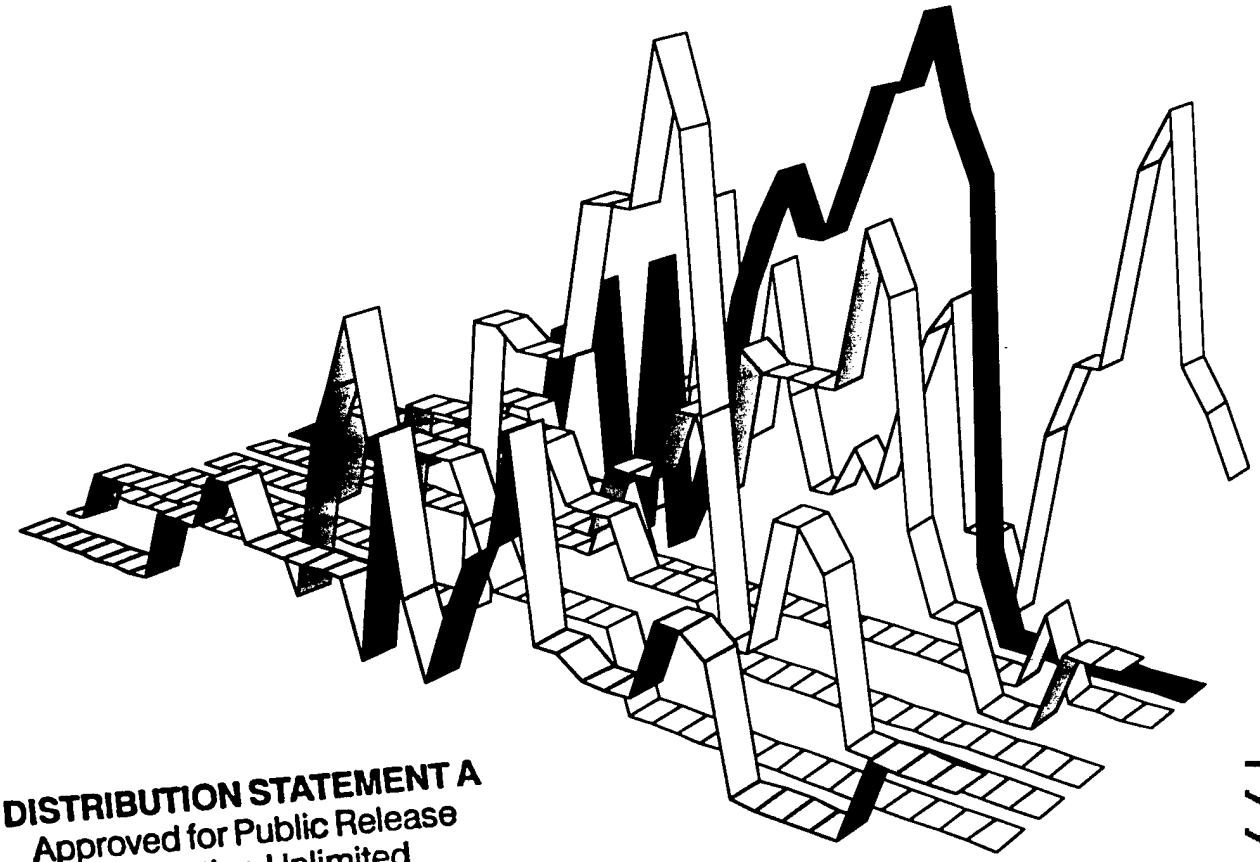
PACIFIC SOFTWARE RESEARCH CENTER
TECHNICAL REPORT

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.9]

Prepared for:
USAF
Electronic Systems Center/AVK

Prepared by:
Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
[illegible]



DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

OREGON
GRADUATE
INSTITUTE OF
SCIENCE &
TECHNOLOGY

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.9]

Prepared for:
USAF
Electronic Systems Center/AVK

Prepared by:
Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
PO Box 91000
Portland, OR 97291

Pacific Software Research Center
Collection of papers from
January 1, 1999 to March 31, 1999

"Formal Verification of Explicityly Parallel Microprocessors"
"Symbolic Simulation of Microprocessor Models using Type Classes in Haskell"
"The Internet As A Medium For Software Engineering Experiments"
"Top-level Refinement in Processor Verification"
"On embedding a microarchitectural design language within Haskell"
"Elementary Microarchitecture Algebra"
"Recursive Function Definition over Coinductive Types"
"DSL Implementation Using Staging and Monads"
"Erasure for termination proofs"

Formal verification of explicitly parallel microprocessors*

Byron Cook, John Launchbury, John Matthews, and Dick Kieburtz
{byron,jl,johnm,dick}@cse.ogi.edu

Version: Fri Mar 5 19:27:54 PST 1999

Abstract. An emerging trend in microprocessor design is to move complexity from a machine's microarchitecture into its instruction-set architecture. This trend will allow compilers to express inter-instruction dependency information that current superscalar out-of-order machines, such as the Pentium III, derive while performing computation. This trend will change the nature of microprocessor verification: The microarchitectural models will become simpler; but their specifications will become more subtle.

This paper explores the implications that this trend will have on microprocessor verification. We develop an explicitly parallel instruction-set architecture motivated by Intel's IA-64 and discuss possibilities for microarchitectural implementations. We then explore correctness criteria for relating microarchitectures to explicitly parallel instruction sets.

1 Introduction

Historically, each generation of microprocessors has been more aggressive than the previous generation in its search and exploitation of instruction-level parallelism [23]. For example, Intel's Pentium III (which is based on the P6 microarchitecture [6, 12]) maintains a graph of 40 instructions, from which it analyzes inter-instruction dependencies and dynamically schedules instructions into execution units.

There is a cost to this sophistication. Complex superscalar out-of-order microarchitectures lead to larger, hotter microprocessors that consume more power [8]. They are difficult to design and debug, and typically have long critical paths, which inhibit faster clock speeds [5]. Some microarchitects feel that the returns are diminishing from their continued investment into the run-time discovery of instruction-level parallelism [25].

A new trend is developing. Intel [13, 14], Hewlett-Packard [13, 19], Compaq [30], Tera [2], Elbrus [9] and others are all extending or designing new instruction-set architectures with constructs for explicit parallelism. These features include predication [1], speculative load instructions [17], and annotations that describe the dependencies between instructions [28].

* This research is supported in part by Intel, the National Science Foundation, the Defense Advanced Research Projects Agency, and Air Force Material Command.

What will these new instruction-sets look like? How will we verify microarchitectures against them? These are the questions that we hope to address. In this paper, we construct a formal semantics for an instruction-set architecture based on publicly available information regarding Intel's new IA-64 [10]. We then develop a clustered microarchitectural design, and discuss its correctness criteria.

2 OA-64: an explicitly parallel instruction set

This section introduces and motivates the emerging style of architecture design through the Oregon Architecture (OA-64) — an explicitly parallel instruction set. OA-64 extends a traditional instruction set in three ways:

Predication allows for the conditional execution of instructions.
Speculative loads are instructions that can be issued before the value they produce is needed without risk of raising an exception.
Parallelism annotations describe the dependencies between instructions.

To see how these features fit into OA-64, look at Fig. 1 which contains an OA-64 code of the factorial function:

- An OA-64 program is a finite sequence of *packets*, where each packet consists of three instructions. OA-64 programs are addressed at the packet-level. That is, instructions are fetched in packets, and branches can jump only to the beginning of a packet.
- Instructions are annotated with *thread identifiers*. For example, the `0` in the `check_s` instruction declares that instructions with thread identifiers that are not equal to `0` can be executed in any order with respect to the `check_s` instruction.
- Packets can be annotated with a fence directive `(FENCE)`, which directs the machine to retire all in-flight instructions before executing the following packet.
- Instructions are predicated on boolean-valued registers. For example, the `check_s` instruction will only be executed if the value of `p5` is true in the current register-file state.

2.1 Calculating regions

Thread identifiers and fences are annotations to express concurrency information about instructions. One useful presentation of this concurrency information is a directed graph whose nodes are sets of threads (which are finite instruction sequences) that occur between fence directives. We call each set of threads a *region*. The general idea is that that an OA-64 machine will execute one region at a time. In this manner, all values computed in previously executed regions are available to all threads in the current region.

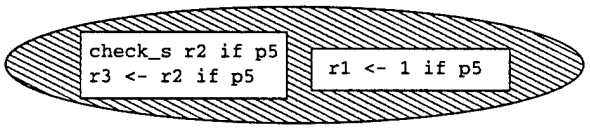
```
101:        check_s r2 if p5 in 0
            r3 ← r2 if p5 in 0
(FENCE)     r1 ← 1 if p5 in 1

102:        p2,p3 ← r2 != 0 if p5 in 0
            r3 ← r2 if p5 in 1
(FENCE)     nop

103:        r1 ← r1 * r3 if p2 in 0
            r2 ← r2 - 1 if p2 in 1
            pc ← 102 if p2 in 2

104:        store 401 r1 if p3 in 3
            pc ← 33 if p3 in 4
(FENCE)     nop
```

Fig. 1. OA-64 implementation of factorial function.

In this section we derive the meaning of the code in Fig. 1 by calculating its regions. We assume that packet 100 issues a fence, and that before entering this code, the machine has loaded a value into register r2 with the speculative load instruction (load_s).

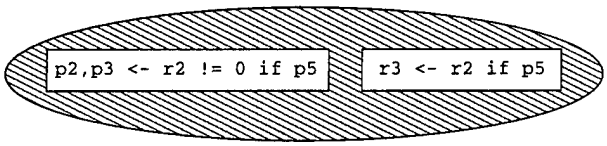
In packet 101, the check_s instruction declares that the machine is about to use the value stored into r2. It is at this point that the machine should raise any exceptions that might have been encountered while speculatively loading data into r2. The first packet also initializes the values of registers r1 and r3. Because r3 depends on the value of r2, the check_s instruction must be executed before writing to r3 — this is expressed by placing the same thread-identifier (0) in the two instructions. The calculation of r1, however, can be executed in any order with respect to the 0 thread.

The fence directive in packet 101 instructs the machine to retire the active threads before executing the following packet. Because both packets 100 and 101 issues fence directives, packet 101 forms its own region:



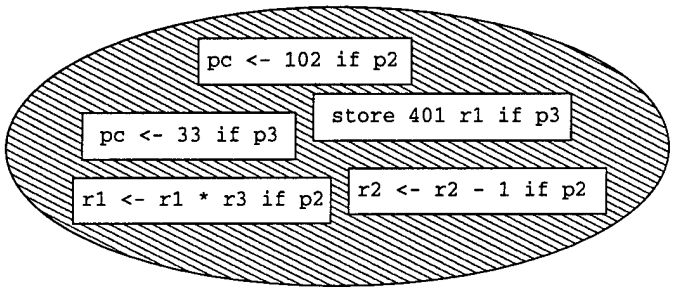
where boxes represent threads. Instructions within a thread must be executed in order. Threads, however, can be executed in any interleaving-order with other threads. Packet 101 forms a region — therefore the machine is required to synchronize the state before executing the next packet.

Because packet 102 is also fenced, it also forms its own region:



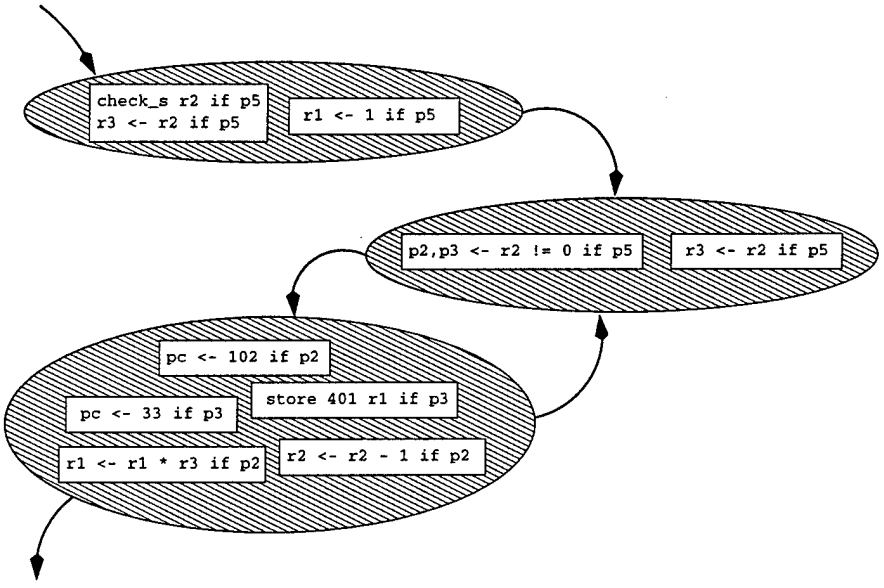
The comparison instruction sets the predicate register p2 to true if r2 is not equal to 0. The value of p3 is set to the negation of p2.

Because packet 103 is not fenced, but packet 104 is, the next region is formed from packets 103 and 104:



This region contains 5 singleton threads. Note that, if both p2 and p3 were true, two threads would write to the program counter (pc) in an arbitrary order. However, because p2 and p3 are the negation of one another, for a given run of the region only one thread will write to pc.

Assignments to the program counter within a region are visible to the machine's fetch mechanism only after a fence directive has been issued. Therefore, a trace of an OA-64 program can be viewed as an infinite path through the finite directed graph formed by regions and their successors:



At first glance, issuing speculative loads and calculating regions may appear strange. However this is precisely the sort of control calculation an out-of-order machine performs while executing a traditional program [25]. For example

- An out-of-order execution core allows instructions following a memory load to execute before retiring the load. The Pentium III temporarily stores completed successors of a load into a content-addressable array until the load is retired, and flushes the array if the load raises an exception.
  The OA-64 code fragment in Fig. 1 uses a `check_s` instruction that checks to see if the previously issued speculative load succeeded before executing the instructions that depend on it.
- A traditional encoding of the factorial function would use a conditional branch in the place of the predicate calculation. A machine with branch speculation might predict that the branch is not taken and issue the instructions in the third packet before calculating the condition. In this case the branch target buffer is acting as the predicate register file.
  The OA-64 program calculates a predicate, issues instructions from both sides of the would-be branch, and in the end only commits the instructions that satisfy the predicate.
- In a traditional instruction set the encoding of the factorial function would leave much of the instruction-level parallelism implicit. The scheduling logic within an out-of-order machine might analyze the register references and discover that the subtract and multiply instructions are not dependent and can be scheduled out-of-order.
  In OA-64, the compiler (or programmer) declares the dependencies between instructions. If the compiler expresses that the subtract and multiply instructions are not dependent, the machine may retire them out-of-order.

3 Semantics of OA-64

In this section we describe a formalization of OA-64 that facilitates the mathematical verification of microarchitectural implementations. The meaning of OA-64 is defined by a set of restrictions on the source program, an initial state, and a transition relation that describes how instructions effect the state.

3.1 Source code restrictions

The following restrictions are placed on OA-64 programs:

- a multiple packet region must always execute at least one branch instruction;
- a branch instruction can only jump to a packet that immediately follows a packet with a fence directive, or to the first packet in the program;
- a program must be a finite sequence of packets;

3.2 Initial state and transition relation

We view OA-64 as a two-level language — the bottom level, or *base-level*, is a vanilla RISC instruction set with support for speculative loads; the top level, or *concurrency-level*, handles predication, thread identifier annotations and the fence directives. The concurrency-level language is used to express dependencies between base-level instructions.

The semantics of OA-64 highlight this perspective by defining a traditional base-level transition relation and a concurrency-level transition relation. The base-level relation $\triangleright$ is defined over instructions and pairs of base-level architectural states — called *base-states* — which represent the state of the register file and memory (the program counter is modeled as the special register `pc` in the register file). The expression:

$$\Delta, w \triangleright \Gamma$$

asserts that instruction w in state Δ can execute and result in state Γ in $\triangleright$. This relation is simply the familiar instruction-set style of relation used in many papers, i.e. $\Delta, (\mathtt{x} \leftarrow \mathtt{y} + \mathtt{z}) \triangleright \Delta[x \mapsto \Delta(y) + \Delta(z)]$

The concurrency-level transition relation $\blacktriangleright$ is defined in Fig. 2 over pairs of concurrency-level architectural states, called *concurrency-states*, which have the form:

$$(P, \Delta, \Sigma)$$

where P is a finite sequence of packets representing the OA-64 program, and Δ is an base-level state. Σ is the state of the region, which is a finite set of finite instruction sequences. Given an OA-64 program, P, the machine's initial concurrency-state is the triple:

$$(P, \mathbf{init}, \emptyset)$$

where **init** is an initialized base-state, and $\emptyset$ is the empty region.

In the initial concurrency-state, or when the machine has completely executed a region, the concurrency-state of the machine will be in the following form

$$(P, \Delta, \emptyset)$$

In this case, the rule **next** (in Fig. 2) states that the machine should use the value of `pc` in the current base-state (Δ) to fetch the next region. The function **region**, when given an OA-64 program and an index, returns the region pointed to by the index. Also, the base-state is updated by incrementing the program counter.

If the region in the current concurrency-state is not empty, then it will have the form

$$(P, \Delta, \langle \ldots, (w \ \mathtt{if}\ p) : ws, \ldots \rangle)$$

where $(w \ \mathtt{if}\ p)$ is the first instruction of an arbitrarily chosen thread[1]. If, in the base-state Δ, the value of p is false then the instruction w is thrown away (rule

[1] We use : as a constructor of lists. In the expression $x : xs$, x is the first element in the list and xs represents the remaining elements

skip in Fig. 2). Otherwise, if p is true in the base-state, then a new base-state Γ is related to Δ and w by $\triangleright$ (rule **execute**).

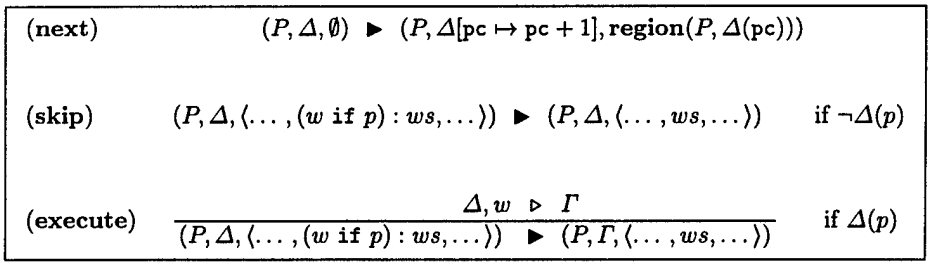


Fig. 2. Concurrency-level semantics of OA-64

4 Columbia: An OA-64 microarchitecture

The advantage of OA-64 is that the microarchitecture can dedicate more of its resources to computation, and less to scheduling. This section presents an outline of a possible microarchitecture for OA-64.

The picture in Fig. 3 is of Columbia, a clustered OA-64 microarchitecture. The machine's execution core is composed of three independent execution pipelines, or clusters. At each cycle Columbia fetches a packet from the **ICache** and feeds it to the clusters. In the case that a packet contains a fence directive, the machine stops fetching instructions until all of the clusters have been flushed.

Fetched instructions travel from the **ICache** to the **Route** unit, where they are routed to one of three pipelined execution clusters based on their thread-identifier (modulus 3). The execution clusters act as traditional in order pipelined execution cores, except that they share a communal register file (**RF**) and data cache (**MCache**). At each clock cycle the clusters calculate how many instructions they can accept on the next cycle. The minimum of these values is sent to the control logic (because all of the instructions in a packet might be routed to one execution cluster). The control logic is also signaled when all of the clusters are in a flushed state.

The fetch logic uses the register file's program counter value. The **Valid** circuit determines, based on whether or not the machine is still servicing a fence directive, if the program counter should be used (i.e. the machine has finished processing a region).

Notice that, in contrast to the large amounts of interconnected state found in superscalar out-of-order models, Columbia's state is smaller and mostly local (i.e. local buffers within execution clusters). This is good news for everyone: The reduced state will be simpler for algorithmic formal verification; and the reduced interaction between components will be good for deduction.

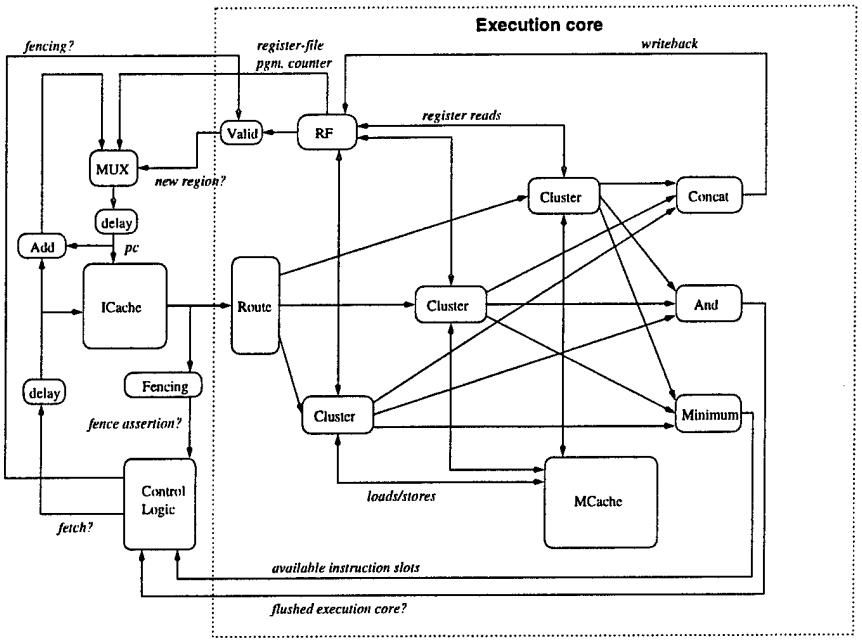


Fig. 3. Columbia microarchitecture — pictured here with three pipeline clusters

5 Verification

Explicitly parallel machines aim to exploit much of the same instruction-level parallelism that superscalar out-of-order machines use — with a twist. They use less hardware, but are more difficult to program. It is therefore natural that the verification of explicitly parallel microarchitectures will be similar to the verification of superscalar out-of-order machines — with a twist. They will be easier to prove correct, but the correctness criteria are more difficult to define.

Assume that, for a given microarchitectural model, ξ_n is a projection representing the machine's region state at time n, and δ_n is the base-state within the microarchitecture. In the case of Columbia, ξ is the contents of the pipelines (and their buffers) and δ equals the contents of the register file and memory cache.

The criteria that we advocate for OA-64 are, for a given program (P), the concurrency-state formed with ξ and δ should infinitely often enter into a reachable concurrency-state defined by the closure of the instruction-set semantics (safety)

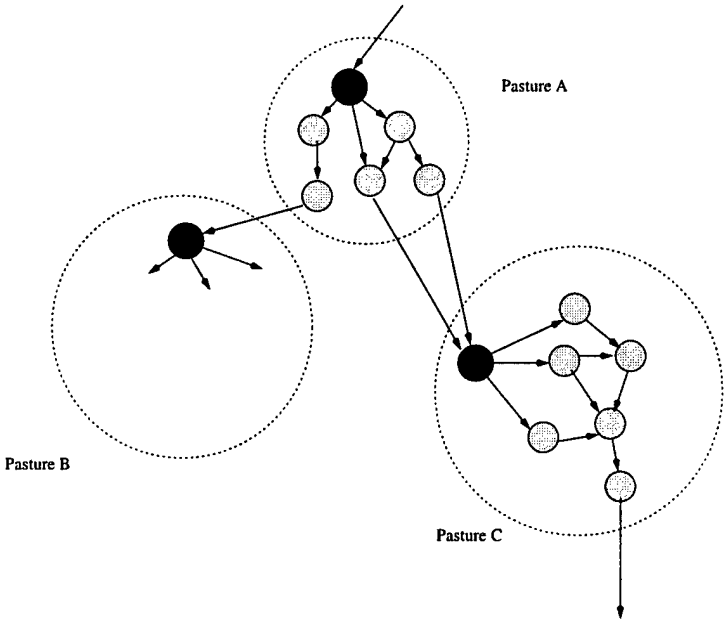
$$\forall n.\exists n'.\ n \leq n' \ \wedge\ (P, \mathbf{zero}, \emptyset) \overset{*}{\blacktriangleright} (P, \delta_{n'}, \xi_{n'})$$

and that the machine infinitely often makes progress in the computation (liveness)

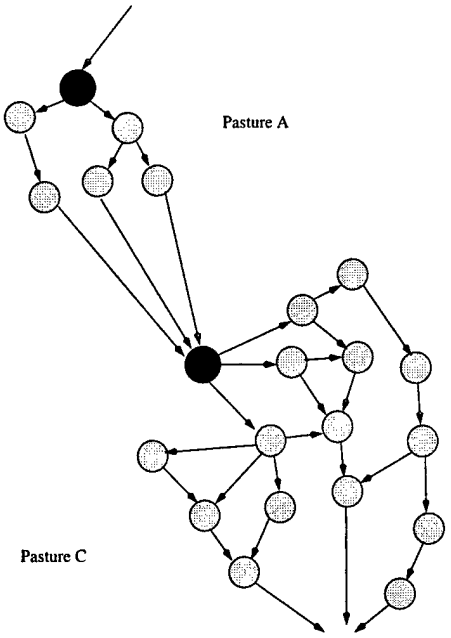
$$\forall n.\ (P, \mathbf{zero}, \emptyset) \overset{*}{\blacktriangleright} (P, \delta_n, \xi_n) \ \Rightarrow\ \exists n'.\ n < n' \ \wedge\ (P, \delta_n, \xi_n) \overset{+}{\blacktriangleright} (P, \delta_{n'}, \xi_{n'})$$

The key here is regions, which declare the existence of *synchronization points* — concurrency-states along the path of execution in which threads have access to the results of computation from previously executed threads. In $\blacktriangleright$, every concurrency-states resulting from a **next** transition is a synchronization point. The formulation of OA-64, coupled with the constraints on the input program, guarantee that regions are always finite. Therefore OA-64 guarantees that the transition **next** will be applied infinitely often.

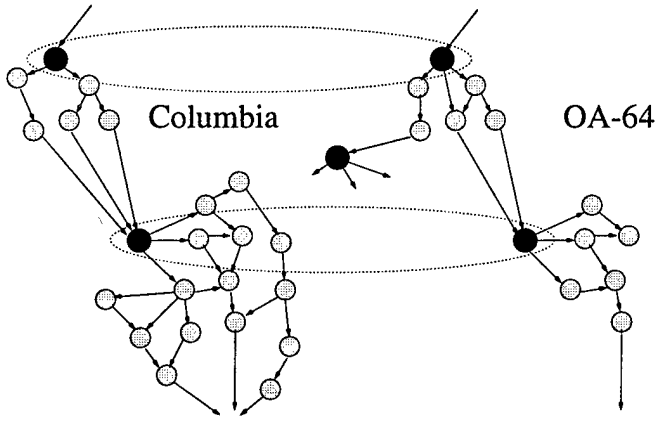
Suppose that, for a given program, the concurrency-state transition graph (based on the region element of the concurrency-state) has the following form



where the black circles are the synchronization points. Also, suppose that the microarchitectural transition graph (based on the value of the microarchitectural thread state ξ) has the form



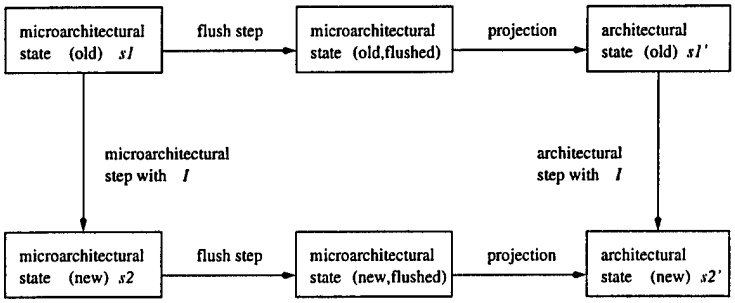
where the black circles represent microarchitectural synchronization points. Between synchronization points the microarchitecture might make more or fewer transitions than the instruction-set architecture. However, when viewing synchronization points, the microarchitecture's transitions are contained by the architecture. The verification problem is then to demonstrate that, when the microarchitecture has reached a synchronization point, the state of the register file and the region that it is executing relates to a reachable concurrency-state in OA-64.



5.1 Adapting pipeline flushing methods

When paired with an inductive proof over the infinite path of regions, the pipeline flushing method [4] for pipeline verification can be adapted to imply the proposed safety property.

In Burch and Dill's formulation, one must prove the commuting square for all possible instructions I:



In the setting of explicitly parallel architectures, we propose letting I range over regions instead of instructions. That is, assume that the microarchitecture begins to execute a region in synchronization point $s1$, and that $s2$ is the next synchronization point resulting from the execution starting at $s1$. Let $s1'$ be the result of flushing and projecting out the architectural state from $s1$, and $s2'$ be the analogous calculation from $s2$. Does there exist a path in $\blacktriangleright$ from $s1'$ to $s2'$?.

A drawback to this formulation is that I no longer has a clear bound (ie. 16 bit instructions). Instead, I is bounded by the size of regions — which is not satisfactory for model checking. In our verification, we made deductive arguments based on the fact that some finite number of cycles after fetching a packet with a fence declaration, Columbia transitions into a synchronization point. We used a symmetry-reduction styled argument to show that, if the microarchitecture fetched an entire region before executing (given sufficient buffering), then that is the same as concurrently executing and fetching that region. The more abstract transition relation calculated from this symmetry argument was then compared to $\blacktriangleright$. The final step was to show that, when the machine has entered into a synchronization point, that it correctly transitions to the next region. This final step was proved using Symbolic Trajectory Evaluation [15]

A useful characteristic of Columbia-like microarchitectural models is that the number and arrangement of clusters doesn't affect the correctness of a microarchitectural design. This is because the transition relation $\blacktriangleright$ allows for any order of evaluation when many threads are trying to write to a shared location in the object state.

No matter how many clusters the execution core employs, so long as the clusters behave analogously to $\triangleright$, the correctness of the execution core outlined in Fig. 3 can be abstractly characterized by the following assertion (certain preconditions have been omitted):

$$(S, \delta_n, (\mathbf{schedule}(\mathtt{fetched}_n, \xi_n))) \overset{*}{\blacktriangleright}_{\{\mathbf{execute,skip}\}} (S, \delta_{n+1}, (\xi_{n+1}))$$

where $\blacktriangleright^*_{\{\mathbf{execute},\mathbf{skip}\}}$ is the closure of the relation $\blacktriangleright$ using only the rules **execute** and **skip**, and **schedule** distributes a packet into a partial region.

Note to reviewers: We're waiving our hands a bit in this section. The statements made in this section are based on a pencil-and-paper proof. We are building a proof in Isabelle which should be done before a camera-ready version of this paper would be due.

6 Related work

The work in this paper is closely aligned in approach with the existing research on the verification of superscalar out-of-order machines [3, 7, 11, 24, 26, 27], all of which use refinement based techniques or flushing (which can be cast as an instance of refinement). In most of these papers, extra information about the dependencies, which OA-64 makes explicit, has been added to the models. For example, Damm and Pnueli construct a non-deterministic data-flow machine that computes the same result as the instruction-set architecture and is refined by a Tomasulo-like transition system. Of course their machines can only execute finite instruction streams that do not contain branches; but their abstract data-flow machine is similar to OA-64.

The instruction set of the Java virtual machine includes facilities for threaded execution. Unfortunatly, the formalizations of the Java virtual machine have, to date, concentrated mainly on type-safety ([22], for example) or have assumed a single-threaded semantics (such as [29]).

Techniques from formal verification have been used to automate the test generation for a dual-issue DLX microprocessor [16] which can be viewed as a simple explicitly parallel machine. The Stanford Validity Checker has been used to show properties of this same processor [18]. However, that paper focuses primarily on the quantifier-free logic of equality with uninterpreted functions and does not go into detail about the properties verified.

7 Future work

The upcoming explicitly parallel instruction-set architectures will take many forms; OA-64 is only one conservative possibility. For example, the real instruction sets might allow sychronization between individual threads; or they might allow branch instructions to take immediate effect on the machine's program counter. Meanwhile, real explicitly parallel microarchitectures will use branch prediction, or even out-of-order clusters to improve performance. The work presented here is conservative in its specification and model. We hope to verify more sophisticated microarchitectures against more realistic instruction sets.

The use of layered transition relations ($\blacktriangleright$ and $\triangleright$) has been invaluable to the understanding and verification of explicitly parallel machines. We hope to generalize this notion, with separate levels for each instruction-set feature: concurrency, predication, speculation, etc. We may find that a particular piece of

a microarchitecture implements a single-level of an instruction-set's semantics; which might allow us to treat the other semantic layers much more abstractly — perhaps as uninterpreted functions.

Letting the I range over entire regions in Section 5.1, while theoretically interesting, makes algorithmic verification difficult. We hope to find other finer-grained approaches (perhaps still based on flushing) that imply correctness.

McMillan's use of symmetry [21] might prove to be useful in the setting of multiple symmetric execution clusters. It should be possible that a small set of cluster configurations could represent all possible cluster configurations. McMillan applied this technique to reduce the number of reservation station and execution unit pairs in his model of Tomosulo's algorithm. He represented all configurations with two reservation station / execution unit pairs — one to represent the active pair, and the other to represent all other pairs.

From $\blacktriangleright$, it might be interesting to develop a reference model and verify more sophisticated OA-64 models against it using the algebraic approach proposed by Matthews and Launchbury [20]. This will involve developing (perhaps non-finite state) circuits that model the characteristics of the instruction set such as predicated execution, speculative execution, etc, and then using algebraic laws to transform the microarchitectural model into a reference machine.

8 Acknowledgments

For their contributions and comments, we would like to thank Mark Aagaard, Todd Austin, and John O'Leary of Intel Corporation; Tim Leonard and Abdelillah Mokkedem of Compaq/Digital Corporation; Mandayam Srivas of SRI; and Tito Autrey, Nancy Day, Sava Krstić, Jeff Lewis, Thomas Nordin, and Mark Shields of OGI.

References

1. Allen, J., Kennedy, K., Porterfield, C., and Warren, J. Conversion of control dependence to data dependence. In *The 10th ACM Symposium on Principles of Programming Languages* (Jan. 1983).
2. Bokhari, S., and Mavriplis, D. The Tera multithreaded architecture and unstructured meshes. Tech. Rep. NASA/CR-1998-208953, NASA/ICASE, 1998.
3. Burch, J. Techniques for verifying superscalar microprocessors. In *33rd annual Design Automation Conference* (Las Vegas, Nevada, June 1996).
4. Burch, J., and Dill, D. Automatic verification of pipelined microprocessor control. In *6th International Conference of Computer Aided Verification* (Stanford, California, June 1994).
5. Case, B. IA-64's static approach is controversial. *Microprocessor Report 11*, 16 (1997).
6. Cook, B., Launchbury, J., and Matthews, J. Specifying superscalar microprocessors with Hawk. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998).

7. DAMM, W., AND PNUELI, A. Verifying out-of-order executions. In *Conference on Correct Hardware Design and Verification Methods* (Montreal, Canada, 1997).
8. DIEFENDORFF, K. Microarchitecture in the ditch. *Microprocessor Report 12*, 17 (1998).
9. DIEFENDORFF, K. The Russians are coming. *Microprocessor Report 13*, 2 (1999).
10. DULONG, C. The IA-64 architecture at work. *IEEE Computer 31*, 7 (1998).
11. FOX, A. C., AND HARMAN, N. A. Algebraic models of superscalar microprocessor implementations: A case study. In *Prospects for Hardware Foundations*. Springer-Verlog, 1998.
12. GWENNAP, L. Intel's P6 uses decoupled superscalar design. *Microprocessor Report 9*, 2 (1995).
13. GWENNAP, L. Intel, HP make EPIC disclosure. *Microprocessor Report 11*, 14 (1997).
14. GWENNAP, L. Intel outlines high-end roadmap. *Microprocessor Report 12*, 14 (1998).
15. HAZELHURST, S., AND SEGER, C.-J. H. Symbolic trajectory evaluation. In *Formal Hardware Verification*. Springer-Verlog, 1997.
16. HO, R. C., YANG, C. H., HOROWITZ, M. A., AND DILL, D. Architecture validation for processors.
17. JOHNSON, D. Techniques for mitigating memory latency in the the PA-8500 processor. In *Hot Chips 10* (Palo Alto, Aug. 1998).
18. JONES, R. B., DILL, D. L., AND BURCH, J. R. Efficient validity checking for processor verification. In *Proceedings of the 1995 International Conference on Computer-Aided Design* (San Jose, 1995).
19. KATHAIL, V., SCHLANSKER, M., AND RAU, B. R. HPL PlayDoh architecture specification: Version 1.0. Tech. Rep. HPL-93-80, Hewlett Packard Laboratories, 1993.
20. MATTHEWS, J., AND LAUNCHBURY, J. Elementary microarchitecture algebra. In *International Conference on Computer-Aided Verification* (Trento, Italy, July 1999).
21. MCMILLAN, K. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).
22. QIAN, Z. A formal specification of a large subset of Java(tm) virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java(tm)*. Springer-Verlog, 1998.
23. RAU, B. R., AND FISHER, J. A. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing 7*, 1 (1993).
24. SAWADA, J., AND HUNT, W. Processor verification with precise exceptions and speculative execution. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).
25. SCHLANSKER, M., RAU, B. R., , MAHLKE, S., KATHAIL, V., JOHNSON, R., ANIK, S., AND ABRAHAM, S. G. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Tech. Rep. HPL-96-120, Hewlett Packard Laboratories, 1996.
26. SHEN, X., AND ARVIND. Design and verification of speculative processors. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998).
27. SKAKKEBAEK, J., JONES, R., AND DILL, D. Formal verification of out-of-order execution using incremental flushing. In *International Conference on Computer-Aided Verification* (Vancouver, Canada, July 1998).

28. SONG, P. Demystifying EPIC and IA-64. *Microprocessor Report 12*, 1 (1998).
29. STEPHENSON, K. Towards an algebraic specification of the Java virtual machine. In *Prospects for Hardware Foundations*. Springer-Verlog, 1998.
30. TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture* (Philadelphia, PA, May 1996).

Symbolic Simulation of Microprocessor Models using Type Classes in Haskell

Nancy A. Day, Jeffrey R. Lewis, and Byron Cook

Oregon Graduate Institute, Portland, OR, USA
{nday, jlewis, byron}@cse.ogi.edu

Abstract. We present a technique for doing symbolic simulation of microprocessor models in the functional programming language Haskell. We use polymorphism and the type class system, a unique feature of Haskell, to write models that work over both concrete and symbolic data. We offer this approach as an alternative to the technique of uninterpreted constants. Compared with previously reported symbolic simulation efforts in theorem provers, the performance of our approach compares favorably, and indeed is several times faster. We illustrate our work with both a simple state-based example and a complex, superscalar, out-of-order, stream-based microprocessor model.

1 Introduction

Symbolic simulation is becoming an important technique for verification of circuits. It can be used by itself for validation of microcode [Gre98] and it is a key ingredient to verification techniques such as symbolic trajectory evaluation [SB95] and Burch-Dill style microprocessor verification [BD94,JDB95]. Symbolic simulation executes a model for multiple data values in a single simulation run. For example, a symbolic program that we discuss in this paper takes the input data x and calculates x^4 (or $x * x * x * x$).

Symbolic simulation of microprocessor models written in the Haskell programming language [PH97] is possible without extending the language or its compilers and interpreters. When symbolically simulating a simple microprocessor model, we achieved performance of approximately 58 300 instructions per second. We describe how Haskell's type class system allows a symbolic domain to be substituted for a concrete one without changing the model or explicitly passing the operations on the domain as parameters. Algebraic manipulations of values in the symbolic domain carry out simplifications similar to what is accomplished by rewriting in theorem provers to reduce the size of terms in the output.

The infrastructure required for using symbolic values and maintaining a symbolic state set is reusable for simulation of different models. We believe the approach presented in this paper may be applied in other languages with user-defined data types, polymorphism, and overloading. However, a key requirement

[0] Draft paper submitted to CHARME'99, 7 March 1999.

is that overloading work over polymorphic types. Few programming languages support this, although a different approach using parameterized modules, as in SML, might also work well. Haskell's elegant integration of overloading with type inference and the clear semantics of the language make it amenable to formal verification.

2 Example

To illustrate our technique, we use the simple, non-pipelined, state-based processor model given in Moore's paper on symbolic simulation [Moo98]. First, we explain the model and demonstrate concrete simulation. Next, we show how using more general types for the data in the model makes it possible to simulate interchangeably concrete and symbolic values. The full source code for this example in Haskell can be found at http://www.cse.ogi.edu/~nday/Papers/sym_sim.html.

2.1 Model

The opcodes of the simple machine are described using a data type:

```
data Op = MOVE Addr Addr
        | MOVI Addr Data
        | ADD Addr Addr
        | SUBI Addr Data
        | JUMPZ Addr Loc
        | JUMP Loc
        | CALL String
        | RET
```

For now, interpret the type names `Addr` (memory address), `Loc` (location), and `Data` as integers.

The machine's visible state is captured by five values: the program counter, the stack pointer, the data memory (modeled as a list, and indexed by integers), the halt signal, and the program. The program is indexed by a name and location because separate routines are stored in distinct memory. Thus, the program counter and elements of the stack consist of both a name and a location. The program consists of names with associated lists of instructions. The machine's state is captured using the following data type[1]:

```
data MachState = ST ((String,Loc), [(String,Loc)], [Data], Bool, Program)
```

The meaning of each instruction is described by individual functions that take a machine state and return a machine state, such as:

```
add a b (ST ((name,loc),stk, mem,halt,code)) =
   mkState ((name,loc+1), stk,
            put a (mem `at` a + mem `at` b) mem, halt, code)
```

[1] In Haskell, list types are represented using square brackets ("[...]").

```
subi a b (ST ((name,loc),stk, mem,halt,code)) =
   mkState ((name,loc+1), stk, put a ((mem `at` a) - b) mem, halt, code)

jumpz a b (ST ((name,loc),stk, mem,halt,code)) =
   if' ((mem `at` a) === 0)
   (mkState ((name,b),stk, mem,halt,code))
   (mkState ((name, loc + 1), stk, mem, halt, code))
```

The semantics of the ADD instruction increase the program counter by one and put the result of the "+" operation on the values in memory locations a and b in memory location a. The function at is an indexing function. In Haskell, to use a regular identifier as an infix operator, you surround it with backquotes, as we did above. The SUBI instruction subtracts the immediate value b from the memory location a. The JUMPZ instruction sets the program counter to the value b if memory location a has the value 0. The operator === is defined to be equality over integers and if' is if-then-else. The function mkState turns a tuple into a state.

The function execute matches opcodes to the semantic functions. For example, execute calls the semantic function for ADD as follows, where s is a state:

```
execute (ADD a b) s = add a b s
```

2.2 Concrete simulation

We can execute the model on particular concrete programs. One of the example programs given in Moore's paper multiplies the value in mem[0] by mem[1] using repeated addition, leaving the result in mem[2], and clearing mem[0]:

```
prog = [ MOVI 2 0,        -- 0, mem[2] <- 0
         JUMPZ 0 5,       -- 1, if mem[0]=0 goto 5
         ADD 2 1,         -- 2, mem[2] <- mem[1] + mem[2]
         SUBI 0 1,        -- 3, mem[0] <- mem[0] -1
         JUMP 1,          -- 4, goto 1
         RET ]            -- 5, return to caller
```

Comments (prefixed by --) on the left describe the meaning of each instruction. Beginning with memory containing the values [7,11,3,4,5] (i.e., mem[0] containing 7 and mem[1] containing 11), and executing the machine for 31 cycles, results in the following memory state: [0,11,77,4,5]. Memory location 2 contains the result of multiplying 7 by 11.

2.3 Overloading: Type classes

We now use the type class system of Haskell to make all the operations that manipulate data be overloaded on both concrete and symbolic data.

In the previous concrete simulation, the type of the function subi is[2]:

[2] In Haskell, a type expression is preceded by a "::" .

```
subi :: Addr -> Data -> MachState -> MachState
```

To simulate symbolic values, we will make it so that the type `Data` can be interpreted at other types than `Int`. We cannot allow `Data` to be any type (i.e., make `subi` polymorphic) because numeric operations are not defined for all types. Alternatively, we could parameterize `subi` by numeric and other operations that are type-specific to the type of data in memory (i.e., symbolic or concrete). This is the approach of Joyce-style representation variables [Joy90], where all the semantic functions are parameterized by what could become a long list of any operations that are type-specific for any opcode.

Our solution is to take advantage of the overloading of operators provided by type classes. A type class groups a set of operations by the type they operate over. The typechecker is able to determine which instance of the operation is being invoked based on the type of its arguments.

The existing Haskell type class `Num` has almost all the operators that we require for data values for this example. In Haskell, a type class definition declares the name of the class and the operations on members of the class. The `Num` class has the following definition:

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  fromInt :: Int -> a
```

Following the first line in this class definition are operators defined on types within this class. Parentheses indicate that the operation is infix. The parameter after the name of the class (`a`) is used to represent a type belonging to this class. The type signatures of the operations are described in terms of this type. The simple machine only requires the use of "+" and "-". The function `fromInt` turns integers into values of type `a`. This capability is very useful when moving to the symbolic domain because it means existing uses of constant integers do not have to be converted by hand into their representation in the symbolic domain – `fromInt` is automatically applied to them.

In Haskell, the type `Int` is declared to be an instance of the `Num` class.

For the `JUMPZ` opcode, the equality operation on data values is also needed. Therefore, we create a new class called `Word` that inherits all the operations of `Num` and includes the operation `===`.

```
class Num a => Word a where
  (===) :: a -> a -> a
```

The use of the operator `=>` in Haskell indicates that the type `a` must be a member of the type class `Num` and therefore the types in `Word` inherit all of `Num`'s operations. The type `Int` is an instance of the type `Word` where the equals operator returns true (1) if the two operand integers are equal and false (0) otherwise. Boolean values are treated as integers.

The type of values in memory now must be elements of the type class `Word`. The types `MachState` and `Program` are parameterized by the type of the memory elements, as in:

```
data MachState a = ST ((String,Loc), [(String,Loc)], [a], Bool, Program a)
```

Opcodes are also adjusted to take immediate values of types in the `Word` class rather than just integers. For example, the type of the `subi` instruction becomes:

```
subi :: Word a => Addr -> a -> MachState a -> MachState a
```

The definition of `subi` does not change.

Concrete simulation of `prog` results in the same state.

2.4 Symbolic simulation of data flow

Once the model has been set up to accept memory values of types within the `Word` class rather than just integers, we can consider an appropriate symbolic domain. Our symbolic domain must include representations of all operations that the model performs on integers. The values of this domain represent syntactic versions of the expressions performed by the machine. An appropriate symbolic domain for this example includes representations for constants (`Const`), symbols (`Var`), and the results of addition and subtraction operations. Using a recursive data type, we describe the values in the symbolic domain as:

```
data Symbo =
      Const Int
    | Var String
    | Plus Symbo Symbo
    | Minus Symbo Symbo
    | Times Symbo Symbo
```

`Plus` and `Minus` will be used to represent the results of addition and subtraction operations on numbers. We include a representation of multiplication (`Times`) because using algebraic laws we can simplify expressions involving addition and subtraction to expressions involving multiplication (Section 2.5).

Next, we create an instance of the `Num` and `Word` type classes providing witnesses showing how the required operations of `Num` and `Word` are implemented for `Symbo`. Fig. 1 shows the instance declarations for `Symbo` that include function definitions (using pattern-matching) for these operations. The last case in the pattern-matching is the default case. We assume for the moment that the operands to the equality operation will only be concrete values.

After providing these instance declarations, all that is necessary to simulate symbolically the program `prog` is to provide symbolic inputs. To calculate $7 * j$, we begin with memory having the values,

```
[7,Var "j",Var "x",Var "y",Var "z"]
```

The result of the program after 31 steps is[3]:

```
[0,j,j + j + j + j + j + j + j,y,z]
```

[3] This output is pretty printed to remove the "Var" and "Const" prefixes.

```
instance Num Symbo where
  Const x + Const y = Const (x + y)
  Const 0 + y       = y
  x + Const 0       = x
  x + y             = x `Plus` y

  Const x - Const y = Const (x - y)
  x - Const 0       = x
  x - y             = x `Minus` y

  Const x * Const y = Const (x * y)
  Const 0 * y       = Const 0
  x * Const 0       = Const 0
  Const 1 * y       = y
  x * Const 1       = x
  x * y             = x `Times` y

  fromInt           = Const . fromInt

instance Word Symbo where
  (Const x) == (Const y)  =  if (x == y) then (Const 1) else (Const 0)
```

Fig. 1. Instance declarations for "Symbo"

This result shows that the sequence of opcodes in the program performs repeated addition resulting in seven additions of 7 being left in memory position 2.

In this example, we only made one input symbolic. If we had made all of memory symbolic, we would not have been able to execute the program because the value in memory location 1 is used to determine if a branch is taken. Because we have not yet defined equality on symbolic values, checking whether a value like `Var "i"` is 0 would cause a run-time error. We extend our example with symbolic branching in Section 2.6.

Symbolic values in memory are used interchangeably with concrete values in memory (e.g., 7) and in the immediate values within the programs (e.g., 0 in `MOVI 2 0`). The function `fromInt` in the `Num` class turns concrete values into symbolic values making this interchangeability possible. Programs running on concrete values and producing concrete output can still be run on the model with the more general types.

2.5 Algebraic simplifications

The symbolic domain must have the same behavior as the concrete domain. For the case of numbers, there are algebraic laws that hold for the concrete domain that can be used to simplify the output of symbolic simulation. For example, `Var x + Var x` is equivalent to `Const 2 * Var x`. These rules can be implemented for the symbolic domain by augmenting the instance declaration

for Symbo with cases that describe the algebraic rules. Two algebraic rules useful for the multiplication program are:

```
Var x + Var y = if (x == y) then Const 2 * Var x
                            else Var x 'Plus' Var y

((Const x) 'Times' (Var y)) + (Var z) =
      if (y == z) then (Const (x+1)) *  (Var y)
                  else (Const x 'Times' Var y) 'Plus' Var z
```

Using these algebraic simplifications, the result of the multiplication program calculating $7 * j$ is `[0,j,7 * j,y,z]`.

These algebraic simplification rules perform the same task as rewriting in a theorem prover.

2.6 Symbolic simulation of control flow

When control values in a program are symbolic, the output of symbolic simulation captures the multiple execution paths that the program could have followed. Memory location 1 is a control value in the program `prog`, because its value is used to determine whether to take a branch or not. To deal with symbolic simulation of control values, we have to extend our idea of a state to include branches representing multiple execution paths. We build this infrastructure on top of the model.

The branching structure will have states at its leaves. The following is a data type for capturing trees of states:

```
data State f a =
       CondS a (State f a) (State f a) |
       Term (f a)
```

The type variable `a` describes the type of the expression that is used to decide which branch to follow. In our symbolic simulation, this type variable is instantiated to `Symbo`. The type variable `f` describes the form of the leaf states. For the simple machine, this will be the type `MachState`. Because `MachState` is parameterized by the type of data in its memory, we use the type expression `f a`, providing the parameter `Symbo` to `MachState`. The data constructor `CondS` represents multiple execution paths that are conditional on the first argument to `CondS`.

To take a step in this symbolic machine, each leaf state must take a step. This may result in new branches in the tree. The function `step_state` is defined over leaf states and invokes the function `execute` described in Section 2.1. Using `step_state`, we can define a function to take steps over our symbolic state:

```
step (Term s) = step_state s
step (CondS a b c) = CondS a (step b) (step c)
```

Next, we need to extend our symbolic domain to include the result of checking for equality over symbolic values. We add one new symbolic value:

```
data Symbo =
    ...
    | Equals Symbo Symbo
```

The definition of equality in the instantiation of Symbo as a member of the Word type class is now extended to:

```
instance Word Symbo where
    (Const x) === (Const y)  =  if (x==y) then (Const 1) (Const 0)
    a === b                  =  Equals a b
```

Finally, we need to have the ability to create branches in the state data structure when conditional jumps are encountered in the program and symbolic data determines which branch to take. The operator `if'` used in the semantics of JUMPZ must be able to sometimes return a terminal state and sometimes return a branch state. We use a multi-parameter type class to capture the behavior of `if'`. A multi-parameter type class allows you to constrain multiple types in a class instantiation. In the case of `if'`, we parameterize the type of the first argument (the deciding value), separately from the type of the the other arguments. The result of the function has the same type as the second and third arguments.

```
class Conditional a b where
    if' :: a -> b -> b -> b
```

For working with concrete states, we need an instantiation that uses the regular if-then-else for concrete values. Since we are treating Booleans as numbers, it checks if its first argument is 1.

```
instance Conditional Int (State f Int) where
        if' a b c = if (a==1) then b else c
```

When the first argument is symbolic, we have a different definition of `if'` that returns a branched state if the argument is symbolic.

```
instance Conditional Symbo (State f Symbo) where
        if' (Const 1) b c = b
        if' (Const 0) b c = c
        if' a b c         = CondS a b c
```

Now without having changed our model, we have the necessary ingredients to simulate symbolic control values[4]. For example, if we run the program prog for 20 steps, with all symbolic values in memory, calculating $i * j$ produces the output found in Fig. 2. In this output, we have included the value of the halt flag for each state. If i is 0, then the result in memory location 2 is 0 and the program has stopped. If i is 1, the result is j and the program has stopped. The last line of the figure is for the case where $i > 4$, so the result will be at least $5 * j$.

[4] The types of the semantic functions change to return a symbolic state but these type changes can be inferred by the typechecker.

```
CondS (i == 0)
  ([i,j,0,y,z],True)
  CondS ((i - 1) == 0)
    ([i - 1,j,j,y,z],True)
    CondS ((i - 2) == 0)
      ([i - 2,j,2 * j,y,z],True)
      CondS ((i - 3) == 0)
        ([i - 3,j,3 * j,y,z],True)
        CondS ((i - 4) == 0)
          ([i - 4,j,4 * j,y,z],True)
          ([i - 5,j,5 * j,y,z],False)
```

Fig. 2. Output of prog after 20 steps with inputs "i" and "j"

3 Symbolic simulation of a superscalar, out-of-order microarchitecture

We are modifying an existing *Hawk* model for a Pentium II-like microarchitecture [CLM98] to use the type class facilities of Haskell for symbolic simulation. This design is a superscalar, out-of-order, with exceptions, pipelined architecture. We are now able to simulate symbolic data flow for programs running on the model.

Hawk is a Haskell-based hardware description language for expressing microarchitecture designs [CLM98,MCL98]. The value of Haskell's higher-order functions and polymorphism are illustrated in this *Hawk* model although we do not have space to describe them in this paper.

Hawk models usually process transactions. A transaction captures the state of an instruction as it progresses through the pipeline. A transaction contains the address of the instruction, its opcode, and the addresses and values of its operands. The transaction may also contain a speculative PC. As the transaction moves through the pipeline, values for input operands and result operands get filled in. The speculative PC is compared to the calculated result of a branch instruction to determine if the pipeline needs to be flushed.

The essential change necessary to use type classes in this design was to modify the values in registers and memory to be of a type belonging to the type class `Num` rather than only integers. This modification also affects the type of addresses because calculations unite the address and value space. Various *Hawk* library devices that manipulate transactions were changed to the more general type.

The `Symbo` data type was used to execute a symbolic program calculating x^4 on this design. Fig. 3 shows our representation of the symbolic DLX [HP96] program. The comments beside each instruction indicate the address where the instruction is placed in memory. The output of simulating a *Hawk* model is a stream of transactions describing the instructions that have been executed. Fig. 4

```
prog_x_4 =
 [ImmIns (ALUImm (Add Signed)) R3 R0 (Var "x")),-- 64: R3 <- R0 + x
  ImmIns (ALUImm (Add Signed)) R4 R0 4),        -- 65: R4 <- R0 + 4
  ImmIns (ALUImm (Add Signed)) R6 R0 1,         -- 66: R6 <- R0 + 1
  ImmIns (ALUImm (Add Signed)) R5 R0 0,         -- 67: R5 <- R0 + 0
                                                -- loop begins here
  RegReg ALU (S GreaterEqual) R1 R5 R4,         -- 68: R4 <- R1 >= R5
  ImmIns BNEZ R0 R1 32,                         -- 69: if (R1==0) then
                                                     goto (70+32/4=78)
  Nop,                                          -- 70: No_op
  RegReg ALU Input1 F2 R6 R0,                   -- 71: F2 <- R6
  RegReg ALU Input1 F3 R3 R0,                   -- 72: F3 <- R3
  RegReg ALU (Mult Signed) F2 F2 F3,            -- 73: F2 <- F2 * F3
  RegReg ALU Input1 R6 F2 R0,                   -- 74: R6 <- F2
  ImmIns (ALUImm (Add Signed)) R5 R5 1,         -- 75: R5 <- R5 + 1
  Jmp J ((-36)),                                -- 76: goto (77-36/4=68)
                                                -- end of loop
  Nop,                                          -- 77: No_op
  RegReg ALU (Add Signed) R1 R0 R6,             -- 78: R1 <- R0 + R6
 ]
```

Fig. 3. Symbolic DLX program for x^4

shows the output of the symbolic x^4 program for 48 cycles. The number on the left is the cycle that the transaction leaves the pipeline. Because this processor is superscalar, multiple instructions may leave the pipeline in one cycle. The program counter is after the cycle number on an output line. The values of the registers used in computation are given in parentheses. If the instruction is a branch, a speculative program counter is included in the transaction.

We are currently extending the *Hawk* library to handle symbolic control paths as well. The key to making this work is to have trees of transactions flowing along the wires instead of just simple transactions. This is similar to how the state in the earlier example became trees of states. However, a Hawk model is stream-based and therefore, does not have explicit access to its state like the earlier example does. Instead of simply having a top-level branching of state, the branching of state must be threaded through the entire model, just as transactions are. This means that most components will need to understand how to handle trees of transactions. We are exploring how to best use a transaction type class to define easily a new instance of transactions that are trees.

Once these modifications to the *Hawk* library have been made, all future models will be able to simulate both concrete and symbolic programs. The symbolic domain presented in this paper is sufficient for many microarchitectures.

```
1:
2:
3:
4:   256:   R3(x) <- R0(0) + x
     260:   R4(4) <- R0(0) + 4
5:   264:   R6(1) <- R0(0) + 1
     268:   R5(0) <- R0(0) + 0
6:   272:   R1(0) <- R5(0) >= R4(4)
7:   276:   PC(280) <- if R1(0) then PC(280) + 32 else PC(280)
                                                    (SpecPC(256))
8:
...
16:
17:  292:   F2(x) <- F2(1) * F3(x)
18:  296:   R6(x) <- F2(x)
     300:   R5(1) <- R5(0) + 1
     304:   PC(272) <- PC(308) + -36                (SpecPC(256))
19:
...
28:
29:  292:   F2(x * x) <- F2(x) * F3(x)
30:  296:   R6(x * x) <- F2(x * x)
     300:   R5(2) <- R5(1) + 1
     304:   PC(272) <- PC(308) + -36                (SpecPC(272))
     272:   R1(0) <- R5(2) >= R4(4)
     276:   PC(280) <- if R1(0) then PC(280) + 32 else PC(280)
...
42:
43:  292:   F2(x * x * x * x) <- F2(x * x * x) * F3(x)
44:  296:   R6(x * x * x * x) <- F2(x * x * x * x)
     300:   R5(4) <- R5(3) + 1
     304:   PC(272) <- PC(308) + -36                (SpecPC(272))
     272:   R1(1) <- R5(4) >= R4(4)
     276:   PC(312) <- if R1(1) then PC(280) + 32 else PC(280)
                                                    (SpecPC(280))
45:
46:
47:
48:  312:   R1(x * x * x * x) <- R0(0) + R6(x * x * x * x)
```

Fig. 4. Stream of transactions resulting from execution of x^4 program

4 Performance

In this section, we consider the performance of our "symbolic simulator". We used the Glasgow Haskell Compiler Version 4.02 [Ghc] for running our tests. Moore provided timing numbers for doing symbolic simulation of the simple machine in the theorem prover ACL2 on a 200 MHz Sun Ultra 2 with 512 MB [Moo98]. Unfortunately, we did not have an equivalent platform available and ran our test cases on a 450 MHz Intel Pentium II with 512 MB memory. Based on SPEC CPU95 integer benchmarks, our platform is roughly two and half times faster than Moore's [SPE].

For concrete simulation, the multiplication program calculating 10 000 $*$1000 for 40 007 cycles took 0.53 seconds with ACL2 at best and 0.54 seconds for us. Here we are comparing Lisp execution to Haskell execution. On a larger concrete test case taking 400 000 cycles for 100 000 $*$ 1000, we achieved approximately 62 200 instructions per cycle (IPC).

In ACL2, the multiplication program with symbolic data flow calculating $1000 * j$ for 4005 cycles took at best 17 seconds with hints and at worst 55 seconds (IPCs of 72 and 235 respectively). Running the same symbolic program took 0.04 seconds for us. When running a much larger test case of 100 000 $* j$ for 400 000 instructions (no branches) we achieved 58 300 IPC.

The multiplication program with symbolic control flow calculating $i * j$ for 2000 cycles took 1.55 seconds, which is approximately 1290 IPC. With branching symbolic programs, printing time is significant.

ACL2 must use its rewrite engine for symbolic simulation, whereas our approach involves executing a functional program. Therefore, we do not suffer a performance penalty for symbolic simulation. Rewriting requires searching a database of rewrite rules and potentially following unused simplifications [Moo98].

5 Related Work

The approach described in this paper is closely related to work on Lava [BCSS98], another Haskell-based hardware description language. They have focused mainly on gate-level descriptions, but Lava has also been used for signal-processing applications. Lava has explored using Haskell features, such as *monads*, to provide alternative interpretations of circuit descriptions for simulation, verification, and generation of code from the same model. A predominant use of a symbolic circuit interpretation in Lava is to produce output for theorem provers. Consequently, their symbolic simulation assigns labels to all subterms and produces a sequence of assertions relating symbolic inputs to outputs. This is like using pointers to build a branching data structure. Because pointers are untyped, this representation loses some of the type information of the expression. Also, a symbolic interpretation must be applied to all parts of the circuit. Our emphasis has been more on building symbolic simulation on top of the simulation provided by the execution of a model as a functional program. In our descriptions of microprocessors, we rely on the standard meaning of function application to connect

components of circuits. We use type classes extensively to choose between a symbolic interpretation or a non-symbolic interpretation of an operation. Both interpretations can be used within the same simulation run. To achieve this flexibility, we build the branching structure into the symbolic domain and use type classes to capture the symbolic operations. The branching structure is threaded through the model. This threading relies on multi-parameter type classes – a recent extension to Haskell.

Symbols in Lisp can be used for symbolic simulation. For example, to generate expressions for input to the Stanford Validity Checker [JDB95], a simple HDL based on Common Lisp is used [BD94]. In this paper, we show how this approach can be done in a strongly-typed, higher-order, functional programming language.

Symbolic simulation can be carried out with uninterpreted constants using rewriting techniques in a theorem prover (e.g., [Joy89,Win90,Moo98,Gre98]) or using more specialized techniques such as symbolic functional evaluation [DJ]. In this form of symbolic simulation, the model is executed over constants of unknown value but the same type as a concrete value. It does not require any changes to the model. However, uninterpreted constants are an element of logic and their use requires the model to be expressed in a logic. Simulation of a logical specification requires special-purpose infrastructure such as rewriting or a means of partial evaluation. Our symbolic domain provides the same effect as uninterpreted constants using a general-purpose programming language.

Type classes provide the infrastructure needed to support the way uninterpreted constants have been used in logical models of microprocessors. Taking advantage of polymorphism in higher-order logic, Joyce first used "representation variables" to bundle operations on data [Joy90]. These operations parameterize both a reference machine and a model of the implementation. The verification effort is valid for any instantiation of these operations. Having an object-oriented flavor, a type class packages the functions of a representation variable in one location. It is not necessary to parameterize all components of the model by type-specific operations. We provide instantiations of the operations of the type class for both concrete and symbolic simulation.

Graph structures such as BDDs and MDGs represent symbolic formulae. Binary decision diagrams (BDDs) [Bry86] are a canonical form for propositional logic. Multiway decision diagrams (MDGs) [CZS+94] are a canonical representation of formulae in many sorted, first-order logic (including uninterpreted functions). In both cases, by iterating a next state relation, these representations can be used to carry out symbolic simulation. BDDs and MDGs are used in decision procedures because of their canonical form. Our form of symbolic simulation for higher-order expressions only calculates terms and does not produce a canonical form. We have not yet characterized the "decidability" of verification efforts involving the symbolic terms we produce.

6 Limitations

Our approach is limited to models expressed as functions, although they may be either state-based as in Moore's simple example or stream-based as in the Hawk Pentium II-like model.

Compared to carrying out symbolic simulation in a logic, in our approach it is necessary to introduce a term structure for the symbolic domain. Our symbols differ from uninterpreted constants in logic in that a programming language has a built-in assumption that elements of user-defined types are distinct. Creating the symbolic term structure requires care because the symbolic domain must have the same properties of the concrete domain. Therefore, the usual equality operation is only defined for the symbolic domain in special cases such as `Var x = Var x`. In this paper, we do not address the issues of how one ensures the symbolic domain has the same properties as the concrete domain.

Our symbolic simulation cannot determine when multiple symbolic decision points conflict and therefore prune impossible execution paths.

Finally, type classes can make fixing type errors a more difficult process. For example, type errors are often masked as missing class instantiations.

7 Conclusion

The most important conclusion of this work is that facilities can be found within some existing programming languages to carry out symbolic simulation of microprocessor models. Using a programming language means symbolic simulation is accomplished by simply running a program. The speed of our method compares well with using rewriting techniques to carry out symbolic simulation. The output of symbolic simulation produced by a model written in a programming language or executable hardware description language can be used as input to verification tools.

Type classes in Haskell make it possible to simulate interchangeably concrete and symbolic values without changing the model. Type classes provide a way to exchange domains of values without requiring explicit parameterization. The class definition specifies the operations on both the symbolic and concrete domains. Algebraic manipulations of values in the symbolic domain reduce the size of the symbolic terms in the output.

The symbolic infrastructure is likely to be reusable for future microprocessor models. Thus, the initial investment in setting up the type classes can be amortized over the ability to simulate symbolically many models.

We intend to continue this work by considering how this form of symbolic simulation can be used in verification techniques. For example, symbolic trajectory evaluation (STE) [SB95] is currently being applied at the bit-level using BDDs as a symbolic representation. To apply STE at a more abstract level a means of symbolic simulation of abstract values, such as the one we have presented, is needed. We intend to investigate the use of STE for microarchitecture verification leveraging off of this work on symbolic simulation.

8 Acknowledgments

For their contributions to this research, we thank Mark Aagaard of Intel; Dick Kieburtz, John Launchbury, and John Matthews of OGI; and Tim Leonard, and Abdel Mokkedem of Compaq. The authors are supported by Intel, U.S. Air Force Material Command (F19628-93-C-0069), NSF (EIA-98005542) and the Natural Science and Engineering Research Council of Canada (NSERC).

References

[BCSS98] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *ACM Int. Conf. on Functional Programming*, 1998.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, volume 818 of *LNCS*, pages 68–79. Springer-Verlag, 1994.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CLM98] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in Hawk. In *Workshop on Formal Techniques for Hardware*, 1998.

[CZS+94] F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. Technical Report RC19676, IBM, 1994. Also *Formal Methods in Systems Design*, 10(1), pages 7-46, 1997.

[DJ] N. A. Day and J. J. Joyce. Symbolic functional evaluation. Submitted for publication.

[Ghc] Glasgow Haskell compiler. http://research.microsoft.com/users/t-simonm/ghc/.

[Gre98] D. A. Greve. Symbolic simulation of the JEM1 microprocessor. In *FMCAD*, volume 1522 of *LNCS*, pages 321–333. Springer, 1998.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Francisco, 1996.

[JDB95] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD*, 1995.

[Joy89] J. Joyce. *Multi-Level Verification of Microprocessor Based Systems.* PhD thesis, Cambridge Comp. Lab, 1989. Technical Report 195.

[Joy90] J. J. Joyce. Generic specification of digital hardware. In *Designing Correct Circuits*, pages 68–91. Springer-Verlag, 1990.

[MCL98] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in Hawk. In *International Conference on Computer Languages*, 1998.

[Moo98] J. Moore. Symbolic simulation: An ACL2 approach. In *FMCAD*, volume 1522 of *LNCS*, pages 334–350. Springer, 1998.

[PH97] J. Peterson and K. Hammond, editors. *Report on the Programming Language Haskell.* Yale University, Department of Computer Science, RR-1106, 1997.

[SB95] C.-J. H. Seger and R. E. Bryant. Formal verification of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, March 1995.

[SPE] SPEC CPU95 results. http://www.specbench.org/osg/cpu95/results/cpu95.html.

[Win90] P. J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, 1990.

The Internet As A Medium For Software Engineering Experiments

Alex Kotov
Oregon Graduate Institute
20000 NW Walker Road
Beaverton, OR 97006, USA
+1 503 690-1121
E-mail: kotov@cse.ogi.edu

ABSTRACT
Empirical software engineering often faces the challenge of large variability of results among individual subjects. Variability can be reduced by using a larger group of subjects, but such group quickly becomes too expensive. Another challenge is finding a group of subjects that is representative of some relevant population of software engineers. This paper explores the potential of using the internet as the medium for software engineering experiments to address the problems of sample size and representativeness.

KEYWORDS
Empirical software engineering, internet experiments, software validation techniques.

1 INTRODUCTION
Software engineering experiments provide information that helps improve software development process. At the same time, experimental findings in software engineering are sparse. One of the problems in software engineering experimentation is high variability between results of individuals that prompts for larger samples. The cost of experiments increases very rapidly with the increase of the number of subjects involved. Finding a large inexpensive pool of subjects for software engineering experiments is not easy. Another problem is related to the sample population, the subjects of the experiment. The goal of an experiment is to make conclusions about some population that is larger than the studied sample (the "target population"). Sample population in software engineering experiments often consists of students attending a certain class, usually taught by the experimenter. "Captive subjects" recruited from a software engineering class usually don't represent any reasonable population. The limited number of subjects and high variance in individual results might be some of the reasons why software engineering experiments often cannot detect a statistically significant difference between the studied phenomena.

2 THE OPPORTUNITY
Wide propagation of the internet in the recent years offered us a new way to address the problems of sample size and representativeness. An experiment can be conducted via the internet making it unnecessary for the subjects to travel to the experiment site. Instead, they would simply connect to the experiment server, view the materials, and perform the required tasks. The experiment server can be made available 24 hours a day so the subjects could participate on their own schedule. Internet-based experiment would require the subjects to have an internet connection and therefore the set of subjects in such experiment can not be considered representative of the entire population. Since the set of individuals that have an internet connection includes most of the students, it can be considered more representative than a set of "captive subjects" from a class.

Internet experimentation offers other important advantages over the traditional classroom-based setup. First, an internet-based experiment is easy to replicate internally or externally. External replication of experiments is important to verify and validate the original results. To replicate such an experiment, researchers would only need to copy the internet-based infrastructure to their own server and inform the participants of the server's location. Second, internet-based experiments can be much easier to study and improve. Even after the the experiment itself is complete, the web-based infrastructure can be left available for everyone to study and learn from. Other researchers could walk through this infrastructure to better understand subjects' experiences long after the original experiment had been completed. Third, the experiment server can be programmed to capture finer details of subjects' work process that often escape investigation in "paper-and-pencil" experiments.

Internet-based experiment can be conducted much faster and at lower cost than a traditional classroom experiment, eliminate the experimenter bias and ensure that all subjects are treated exactly the same. It also

allows the subjects to remain completely anonymous.

3 THE PROBLEM

Before internet-based experiments become a standard tool of empirical software engineering, research is required to demonstrate that such experiments can produce valid results. Internet-based experiment setup removes some of the threats to validity of an experiment such as experimenter bias and peer pressure, but it can also introduce new ones. Some of the potential threats are: *Control:* The degree of experimenter's control over subjects on the internet is much less than in a classroom experiment, and violation of the rules of the study by subjects would be hard to detect. *Commitment:* In an internet experiment subjects may feel detached and less committed to the study than in a classroom experiment. *Retention:* With the amount and diversity of information available on the internet, subjects will be tempted to leave the experiment site and "surf" somewhere else. *Local conditions:* Subjects may participate in an internet experiment from a location that does not allow them to concentrate. *Technical:* They can experience problems with their computers or internet connections.

There are other factors that can potentially affect the validity of an internet-based experiment. With a diverse participant base, we can expect most of these factors to be randomly distributed so that they will introduce "noise", instead of a bias, into the experimental results. A large number of participants would allow us to collect enough data to filter out the "noise".

4 METHODOLOGY

At the this time we don't have a good understanding of all factors influencing the results of "internet" participants. To overcome this problem, we can aggregate these factors into a single "internet factor" and start by studying how this factor affects the results of internet-based experiments. We can assess the internet factor in two ways. We can quantify it by investigating its effect on the experimental results. To do this, we will design a study that includes the internet factor as an independent variable. We can also try to better understand this factor qualitatively by identifying its major parts.

Quantitative Analysis

We will start the quantitative analysis of the internet factor by designing a study that includes this factor as an independent variable. The study will use two groups of subjects. One group will be recruited locally among students of computer science classes and computer professionals (the "local" group). The other group will be recruited on the internet using postings in usenet newsgroups and submissions to WWW search engines (the "internet" group). Both groups will perform the same set of tasks using the same internet-based infrastructure. The subjects from the local group will perform the tasks using computer terminals in the experiment lab. The subjects from the internet group will perform the same tasks remotely without making the trip to the experiment site. Subjects' task will be to apply different validation techniques to a set of small programs. The techniques selected are "functional testing" (validation without access to source code) and "structural testing" (validation with access to source code). The validation technique will become the second independent variable. The set of programs was created by Kamsties and Lott [1] and later used in a replication by Wood et al. [2].

The difference in performance between local and internet subjects using the same validation technique will allow us to quantify the internet factor. It is possible that the internet factor will introduce a bias, shifting the performance of the internet group up or down for both techniques. It can also change the effect size. Analysis of variance will be used to determine the effect of the internet factor and its interaction with different testing techniques.

Qualitative Analysis

To perform qualitative analysis of the internet factor we can observe the behavior of subjects from local and internet groups by studying the information recorded by the experiment server. It is possible that subjects from the internet group will be more impatient, less attentive, and less likely to read instructions. They may jump from page to page more quickly and make more mistakes. Another way to collect qualitative information is to ask all subjects to fill out a questionnaire at the end of the study. The questionnaire will ask about subject's physical conditions, connection speed, possible interruptions, or other factors that may have affected their performance, and offer a space to provide feedback.

5 CONCLUSION

The internet presents an inviting opportunity to conduct "distributed" experiments that may address some of the most common problems of empirical software engineering: sample size and representativeness. However, research is required to demonstrate feasibility and validity of such experiments by studying, both quantitatively and qualitatively, the factors that affect their results.

REFERENCES

[1] E. Kamsties and C. Lott. An empirical evaluation of three defect-detection techniques. *In Proc. ESEC '95*, LNCS Nr. 989, p. 362-383.

[2] M. Wood et al. Comparing and combining software defect detection techniques: A replicated empirical study. *In Proc. ESEC '96*, LNCS Nr. 1301, p. 262-277.

Top-level Refinement in Processor Verification

Sava Krstić, Byron Cook, John Launchbury, and John Matthews

Oregon Graduate Institute
{krstic, byron, jl, johnm}@cse.ogi.edu

Abstract. We provide a framework for the specification and verification of high-performance processors. As an example, we give a high-level specification and correctness proof for a processor that uses speculation, register renaming, superscalar out-of-order execution, and resolution of memory dependencies. The specifications of its three concurrently operating units are very general and can be refined independently, so that our proof covers a whole family of microarchitectures. Abstract treatment of data, representation of on-the-fly instructions as transactions, and a history table containing the full information of a processor's run are the main features of the proof.

1 Introduction

A variety of formal verification tools are now in use in various phases of hardware design; [2, 8, 17] are but a few notable examples. At the microarchitectural level, however, the real use of verification is limited, mostly due to the immaturity of the available techniques. Indeed, proving the correctness of a combination of aggressive strategies to resolve inter-instruction dependencies is extremely difficult. Still, it is an important verification aspect because microarchitectural defects can impact a large fraction of the design and so are hard to fix. Engineers close to current processor design teams inform us that designers purposefully forgo promising optimizations because they cannot guarantee the optimizations preserve correctness.

Following the top-down approach, we address the question of specifying and verifying processors at a high level. On a worked out example, we show how to abstract the specification as much as possible in order to clearly and concisely specify a complex microarchitecture with the following package of features: speculation, register renaming, superscalar out-of-order execution with in-order retirement, and resolution of memory dependencies. We present only the essentials of the microarchitecture, just enough to make the correctness proof possible. The lower-level details are left to further refinement.

Our example is based on an executable processor model expressed using *Hawk*, a specification language with stream transformer semantics [7, 15]. This example microarchitecture is close to Intel's PentiumPro [10] and AMD's K6 [20]. It is partitioned into three major units for which we provide independent axiomatic specifications. We show that the visible output computed by this microarchitecture is equivalent to that of a simple reference machine implementing

the instruction set architecture. This approach exhibits a very desirable form of modularity where the three units can be independently refined further without affecting global correctness. Moreover, since the units are to a large extent underspecified, our proof covers a whole family of microarchitectures that can significantly vary in implementation details.

To write the specifications and organize the proof, we use a small number of concepts and structures of a general nature. For example, our correctness criterion can be used for any model with in-order retirement. Next, *transactions* (a formalized notion of partially computed instructions) seem to be just the right microarchitectural abstraction that provides uniformity in the description of the data path. Transactions come with a natural partial order (progress in computation of an instruction) that enhances their expressiveness and can be effectively used in reasoning. The proof itself revolves around a *history table* which contains all crucial information about a single run of a processor.

After a brief discussion of related work, the rest of the paper is organized by sections, as follows: we specify a reference machine, introduce transactions and (informally) our processor model, describe the correctness criterion, explain the history table and the structure of the proof, and give formal specifications of the three processor components. The full definition of the history table and a proof of the correctness theorem are relegated to the Appendix.

2 Related Work

The complexity of verified processor models described in the literature varies, largely in connection with the level of proof automation. Highly automated methods show a promising trend of consistent increase of applicability, including impressive recent proofs of out-of-order execution [5, 16]. Still, the models verified by these methods are rather limited. This paper belongs to the other end of the spectrum: our processor model is one of the most complex, but at the price of having been specified in a rather unconstraned mathematical style, and verified by a pencil-and-paper proof. The same can be said of the work of Arvind and Shen [4], whose appealing processor model is defined as a term-rewriting system. While our specifications allow refinement in the most obvious sense, it is not clear how the correctness result of [4] that relies on being able to apply the rewrite rules in any order would translate to a lower-level implementation that lacks that property.

With Pnueli and Arons [18] we share the insistence on maximal abstraction and modularity stemming from specifying the processor as a simple composition of concurrent subsystems. There is also some similarity in the correctness criterion, based on the idea of refinement. Their model, however, assumes a restricted instruction set, without branches and memory instructions.

The correctness criterion adopted in most processor verification papers is the "commutative diagram" condition of Burch and Dill [6], or some version thereof (*cf.* [4, 12, 14, 19]). Along with [18], we avoid dealing with explicit synchronization and abstraction functions that match the states of the verified processor

with the states of the reference machine. Instead, our criterion requires that the two sequences of retired instructions arising from running the same program on the two machines are equivalent.

Dealing with memory instructions combined with out-of-order execution has only recently come into the scope of processor verification efforts; *cf.* [4, 12, 19]. Our execution unit allows multiple refinements with arbitrarily sophisticated treatment of memory operations (load bypassing, for example).

A remarkably detailed model, including a treatment of exceptions, is verified by Sawada and Hunt [19] using a methodology which has many similarities to our work. The key structure they use, the *Microarchitectural Execution Trace Table*, contains entries that are much like our transactions. This table represents the current computational state of the processor like a row of our history table does. A global invariant relates the table with the corresponding microarchitectural state. Since it references most of the state elements, this invariant presents a difficult proof obligation, which unfortunately is only briefly discussed in [19].

Our paper promotes hierarchical verification by providing a very general and non-deterministic model and a straightforward reduction to verification of components. At this level, the assume-guarantee style takes a simple form: all that the components assume of the environment are type-correct values on their input wires; *cf.* [11].

3 Standard machine (ISA)

Our reference model is an abstract *standard machine*, defined as a state machine whose states consist of values for the program counter, register file and memory. Most of the common instruction set architectures are instances of it when we ignore the treatment of external exceptions.

Definition 1. *Given a state* $(\mathtt{pc}, \mathtt{rf}, \mathtt{mem})$*, the standard machine (executing a fixed program* pgm*) makes a transition to the state* $(\mathtt{pc}', \mathtt{rf}', \mathtt{mem}')$ *defined by the following set of equalities.*

$$
\begin{aligned}
I &= \mathtt{pgm}(\mathtt{pc}) \\
(\mathtt{opcode}, \mathtt{rSources}, \mathtt{rDest}) &= \mathtt{decode}(I) \\
\mathtt{rOps} &= \mathtt{rf}(\mathtt{rSources}) \\
(\mathtt{mSource}, \mathtt{mDest}) &= \mathtt{getAddr}(\mathtt{opcode}, \mathtt{rOps}) \\
\mathtt{mOp} &= \mathtt{mem}(\mathtt{mSource}) \\
(\mathtt{pc}', \mathtt{rRes}, \mathtt{mRes}) &= \mathtt{compute}(\mathtt{pc}, \mathtt{opcode}, \mathtt{rOps}, \mathtt{mOp}) \\
\mathtt{rf}' &= \begin{cases} \mathtt{rf}[\mathtt{rDest} \mapsto \mathtt{rRes}] & \textit{if } \mathtt{rDest} \in \mathbf{Reg} \\ \mathtt{rf} & \textit{if } \mathtt{rDest} = () \end{cases} \\
\mathtt{mem}' &= \begin{cases} \mathtt{mem}[\mathtt{mDest} \mapsto \mathtt{mRes}] & \textit{if } \mathtt{mDest} \in \mathbf{Addr} \\ \mathtt{mem} & \textit{if } \mathtt{mDest} = () \end{cases}
\end{aligned}
$$

The function `decode` extracts the opcode, source registers and the destination register from an instruction. The function `getAddr` computes the addresses

mSource for loads and mDest for stores. Finally, the results of compute are the new value for the program counter and the values to be written back to the register file or memory.

The standard machine is totally data-insensitive. It uses abstract basic types **IAddr**, **Instr**, **Opcode**, **Value**, **Reg** and **Addr**, and the rest is typed as follows:

$$\begin{aligned}
&\texttt{pc}: \mathbf{IAddr},\ \texttt{pgm}: \mathbf{IAddr} \to \mathbf{Instr},\ \texttt{rf}: \mathbf{Reg} \to \mathbf{Value},\ \texttt{mem}: \mathbf{Addr} \to \mathbf{Value} \\
&\texttt{decode} : \mathbf{Instr} \to \mathbf{Opcode}, \mathbf{RegSeq}, \mathbf{Reg}^\sharp \\
&\texttt{getAddr} : \mathbf{Opcode}, \mathbf{ValueSeq} \to \mathbf{Addr}^\sharp, \mathbf{Addr}^\sharp \\
&\texttt{compute} : \mathbf{IAddr}, \mathbf{Opcode}, \mathbf{ValueSeq}, \mathbf{Value}^\sharp \to \mathbf{IAddr}, \mathbf{Value}^\sharp, \mathbf{Value}^\sharp
\end{aligned}$$

where we follow the convention to write product types using commas and function types using arrows. The notation $\mathbf{Type}^\sharp$ is a shorthand for the sum type $\mathbf{Type} + \{()\}$, where the element () indicates a value that does not need computation. For example, the first component of the result of getAddr is () unless the first argument is the opcode of a load instruction. Note that our definition allows a single instruction to have the combined behavior of a branch, alu-instruction, load and store, if desired. Particular instructions may of course choose to only implement a subset of this functionality.

4 An example processor

When reasoning about the execution process of complex processors one normally thinks of instructions as entities that come into being at a certain cycle and evolve thereafter. *Transactions* formalize this notion of partially computed instructions. Informally, a transaction is a package of information which (directly or indirectly) contains the identity of the unique (static) instruction it is associated with plus various data extracted from the processor's state that are relevant for the execution of that instruction.

Guided by the standard machine specification, we define a *standard transaction* as a record with the following eleven fields:

instr	: **Instr**	rDest	: $\mathbf{Reg}^\sharp$
opcode	: **Opcode**	mSource, mDest	: $\mathbf{Addr}^\sharp$
rSources	: **RegSeq**	npc	: **IAddr**
rOps	: **ValueSeq**	mOp, rRes, mRes	: $\mathbf{Value}^\sharp$

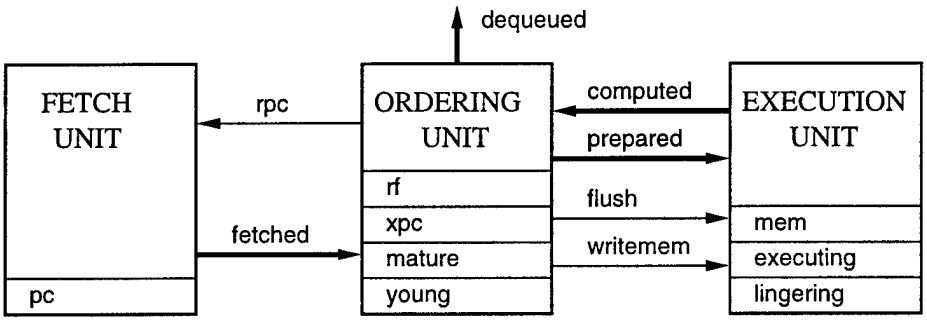
We assume that all our basic types contain a value $\perp$, indicating an uncomputed value. We will also use the notation $\mathsf{rOp}_i(T)$ for the i^{th} member of the sequence $\mathsf{rOps}(T)$. The functions decode, getAddr and compute treat $\perp$ as an argument in a lazy fashion: a component of their result is $\perp$ only if some crucial arguments needed for computation of that result are $\perp$.

A natural idea, introduced in [3] and paradigmatic for the *Hawk* specification language [15], is to use transactions as a unifying concept in microarchitectural specifications. Transactions are passed along wires and manipulated by processor

components. In addition to the above standard fields, any specific microarchitecture adds fields appropriate for the description of its execution algorithm. Our example processor adds five new fields: the *instruction address* addr, the *speculative next program counter* spc, the *name* (alias) name, the *register providers* rProvs and the *most recent store* mrSt:

$$\begin{array}{llll} \mathsf{addr}, \mathsf{spc} & : \mathbf{IAddr} & \mathsf{rProvs} & : \mathbf{NameOptSeq} \\ \mathsf{name} & : \mathbf{Name} & \mathsf{mrSt} & : \mathbf{NameOpt} \end{array}$$

The fields rProvs and mrSt will record dependencies among instructions. Here $\mathbf{NameOpt} = \mathbf{Name} + \{\text{NONE}\}$ is the type of an optional name field, where NONE serves to indicate the lack of dependency.



`pc, xpc`	: **IAddr**	`rpc`	: $\mathbf{IAddr}^\sharp$
`rf`	: **Reg** $\rightarrow$ **Value**	`fetched, dequeued, prepared`	: **TransSeq**
`mature, young`	: **TransSeq**	`computed`	: **TransSet**
`mem`	: **Addr** $\rightarrow$ **Value**	`flush, writemem`	: **Bool**
`executing, lingering`	: **TransSet**		

Fig. 1. Top-level specification with the types of wires (right) and state components (left). Thick wires represent transaction sets or sequences. At each cycle, units update their state and output wires depending on the values on their input wires and state elements at the previous cycle.

The processor consists of three major units and seven wires as depicted in Fig. 1. The *fetch unit* provides multiple instructions at each cycle. This unit outputs along the `fetched` wire transactions with filled in fields instr, addr and spc. The fetching of instructions begins at the address pc if the current value of `rpc` (requested program counter) is (); otherwise `rpc` is used. The fetching proceeds by unconstrained speculation.

The *ordering unit* maintains the sequential programming model of the ISA by using a queue made by concatenating the sequences `mature` and young (Fig. 2). It takes a prefix of the sequence `fetched` to form a transaction sequence enqueued to be added to the back of the queue. The transactions of `fetched` that do not

belong to the chosen prefix are discarded. Each transaction added to the queue gets its name field filled in, unique in the queue. The mature part of the queue corresponds to transactions already sent to the execution unit. Transactions in prepared are taken from the beginning of the young part of the queue and possibly also from enqueued; they all have their rOps, rProvs and mrSt fields filled in. The elements of rOps obtain values from rf when there is no dependency on previous transactions; if there are dependencies, they are recorded in the elements of rProvs, which contain the names of the transactions that will provide the appropriate values when computed. The field mrSt contains the name of the last preceding store in the queue; it is used only by loads and stores for future resolution of dependencies among them. The mature part of the queue is updated by transactions arriving along the computed wire, then a prefix of the resulting sequence consisting entirely of complete transactions is retired, that is, sent along the dequeued wire while updating rf. When a retired transaction is a mispredicting branch, then the queue is emptied, the Boolean wire flush is asserted and rpc set equal to the address of the last retired transaction. The wire rpc is also given a non-trivial value when not all fetched transactions are enqueued. In this case the rpc is set to the spc of the last enqueued transaction.

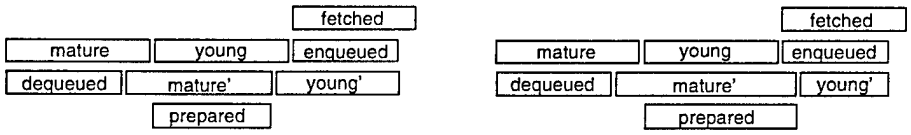


Fig. 2. Two possible scenarios for the relationship between transaction sequences involved in a transition of the ordering unit. The inputs are fetched, mature and young, and the outputs are dequeued, prepared, mature′ and young′. The sequences are aligned so that if two transactions are on the same vertical line, then the higher one is less than or equal to the lower (in the progress ordering defined below).

The *execution unit* is an out-of-order component that computes the results rRes and mRes of transactions contained within it and determines which of these transactions are mispredicting (by computing npc for each and comparing it with spc). It may also execute a memory store if the value on the wire writemem indicates that it is right time to do so. A number of completed transactions are sent out along the computed wire, while placing them in the set lingering, where each of them will remain intact until the moment when an equally named transaction comes along the prepared wire and takes its place. When a transaction is sent to computed (or sooner), the values in its result fields are forwarded to all other transactions in executing. There are no requirements on the number of transactions executed at each cycle and the only requirement on the order of their execution is that the data-flow order is respected.

5 Correctness criterion

One can slightly extend the definition of the standard machine so that at each cycle it outputs a complete transaction (corresponding to the instruction completed at that cycle). A run of the standard machine then defines a sequence of "retired" transactions from which the corresponding sequence of states of the standard machine can easily be reconstructed.

A transition of a complex processor cannot, in general, be associated with a unique transaction, but with a sequence, possibly empty, of transactions retired on that transition. So, suppose P is a processor and denote by ρ_n the sequence of transactions retired by P on its n^{th} cycle. Concatenating these sequences we obtain $\rho_\infty = \rho_1\rho_2\cdots$. Replacing every transaction in ρ_∞ with the corresponding standard transaction (which amounts to ignoring its "non-standard" fields), we obtain a sequence of standard transactions ρ_∞^{std}, which, if P does implement the standard machine, should be identical to the appropriate execution sequence of the standard machine. This gives us the following correctness criterion.

Definition 2. *A processor P is* correct with respect to the standard machine *if for any given program* pgm *and a state σ_0 of the standard machine, there exists an initial state of P such that the execution of* pgm *on P produces a sequence of retired transactions ρ_∞ with the associated sequence ρ_∞^{std} equal to the execution sequence defined by the program* pgm *and the initial state σ_0.*

The notion of the execution sequence is made precise below, after a brief elaboration of the type of transactions.

5.1 The progress ordering of transactions

We define the *progress ordering* $\preceq$ on the set of transactions so that $T_1 \preceq T_2$ will mean that T_2 is a computationally more advanced ("closer to retirement") version of T_1. The relation $\preceq$ is the product of 16 partial orders (all denoted $\preceq$)—one for each record component. These component orders are defined as follows. For each basic type (including **Name**), we make $\perp$ the smallest element and all other elements, including (), incomparable. In **NameOpt**, NONE is the largest element. Finally, two sequences are comparable if and only if they have the same length and the elements of one of them are all less than or equal to the corresponding elements of the other.

The partial order just introduced allows us to define the notion of intrinsic consistency of transactions. Intuitively, a transaction is consistent if the contents of its fields do not contradict any of the equations occurring in the definition of the standard machine. Of these equations, the ones that do not involve the components of the machine state (program counter, register file and memory) give rise to consistency criteria:

$$\langle \mathsf{opcode}(T), \mathsf{rSources}(T), \mathsf{rDest}(T)\rangle \preceq \mathtt{decode}(\mathsf{instr}(T))$$

$$\langle \mathsf{mSource}(T), \mathsf{mDest}(T)\rangle \preceq \mathtt{getAddr}(\mathsf{opcode}(T), \mathsf{rOps}(T))$$

$$\langle \mathsf{npc}(T), \mathsf{rRes}(T), \mathsf{mRes}(T) \rangle \preceq \mathtt{compute}(\mathsf{addr}(T), \mathsf{opcode}(T), \mathsf{instr}(T), \mathsf{rOps}(T), \mathsf{mOp}(T))$$

By definition, a transaction is *consistent* if its fields satisfy these inequalities. We define **Trans** to be the set of all consistent transactions. Note that consistency of a transaction depends entirely on the contents of its "standard" fields and that all strictly increasing chains in the poset $(\mathbf{Trans}, \preceq)$ are of finite length.

Maximal transactions with respect to the ordering $\preceq$ will be called *complete*; a transaction is complete if none of its fields is $\perp$, and mrSt and all component fields of rProvs are NONE.

5.2 Execution sequences

For every transition of the standard machine there is an associated complete standard transaction. To define it, just use the left-hand sides of the equations in Definition 1. Thus, together with every run of the standard machine, one can consider the corresponding transaction sequence $\langle T_1, T_2, \ldots \rangle$, where T_i corresponds to the i^{th} transition. Characterizing properties of such sequences are collected in Definition 3 below.

If τ is a (finite or infinite) sequence of transactions or standard transactions and T a transaction in τ, we define the i^{th} *register provider* of T to be the transaction U of τ which precedes T and has the property that $\mathsf{rSource}_i(T) = \mathsf{rDest}(U)$, while $\mathsf{rSource}_i(T) \neq \mathsf{rDest}(V)$ for all transactions V between U and T. Similarly, we define U to be the *store provider* of T if T is a load and U is the last store among the transactions that precede T in τ and satisfy $\mathsf{mSource}(T) = \mathsf{mDest}(U)$.

Definition 3. *An infinite sequence $\tau = \langle T_1, T_2, \ldots \rangle$ is an* execution sequence *corresponding to the program* pgm *and the initial state $(\mathtt{pc}_{init}, \mathtt{rf}_{init}, \mathtt{mem}_{init})$ if every T_m is a complete transaction and*

$$\mathsf{instr}(T_m) = \begin{cases} \mathtt{pgm}(\mathtt{pc}_{init}) & \text{if } m = 0 \\ \mathtt{pgm}(\mathsf{npc}(T_{m-1})) & \text{if } m > 0 \end{cases}$$

$$\mathsf{rOp}_i(T_m) = \begin{cases} \mathsf{rRes}(T_k) & \text{if } T_k \text{ is the } i^{\text{th}} \text{ register provider for } T_m \text{ in } \tau \\ \mathtt{rf}_{init}(\mathsf{rSource}_i(T_m)) & \text{if } T_m \text{ does not have an } i^{\text{th}} \text{ provider in } \tau \end{cases}$$

$$\mathsf{mOp}(T_m) = \begin{cases} \mathsf{mRes}(T_k) & \text{if } T_k \text{ is the store provider for } T_m \text{ in } \tau \\ \mathtt{mem}_{init}(\mathsf{mSource}(T_m)) & \text{if } T_m \text{ does not have a store provider in } \tau \end{cases}$$

6 History Table (Structuring the proof)

Reasoning about the execution of processors can be conveniently organized around a *history table*. Two simple observations are behind its definition. First, if $I_1, I_2, \ldots$ is the sequence of instructions considered by the processor during a run, then each transaction T found anywhere in the processor at any time is associated with a unique fetched instruction I_j; we say that j is the *ordinal* of T. The second observation is that there are only finitely many essentially different

execution patterns for an instruction and that one can define a finite transition diagram describing those patterns. Each node of this *transaction flow diagram* Γ corresponds to a distinguished "pipeline stage" and will be called a *status.*

A history table is defined for every run of the processor. At the n^{th} row and the i^{th} column of the table one finds a pair $H_n^i = (T, X)$, where T is the transaction that represents the state of computation of I_i at the n^{th} cycle and X is the status of that computation. Formally, H_n^i is defined in terms of the set of transactions with ordinal i which are present in the processor at the n^{th} cycle, and the values of "control" variables at that cycle; normally, T is the maximal of those transactions and the status X corresponds to the set of locations in which they are found.

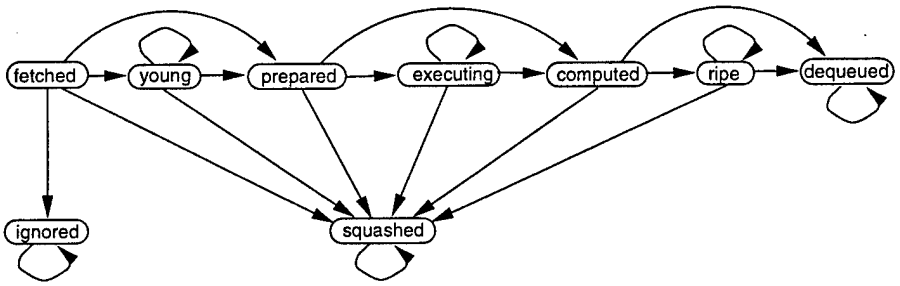


Fig. 3. Transaction flow diagram Γ. The transitions to **squashed** occur only when **flush** = TRUE.

For our example processor, Γ is given in Fig. 3. The top row represents the execution patterns of successfully completed instructions. Looping at young means waiting to be sent to the execution unit; the loops at `executing` and `ripe` have similar meaning. The status `ripe` corresponds to the set of complete transactions contained in `mature`. The final statuses `ignored` and **squashed** are for transactions aborted because of the overflow in the ordering unit (inability to enqueue all fetched transactions) and misprediction, respectively.

The rows of the history table are finite; the length of the n^{th} row is equal to the total number of fetched instructions in the first n cycles. All columns stabilize: for each i, we have $H_{n+1}^i = H_n^i$ for all large n. This follows since both **Trans** and Γ are posets in which strictly increasing chains are finite. We define the *limit row* H_∞ as the sequence of the limit values of columns: $H_\infty^i = \lim_n H_n^i$.

For any $n \leq \infty$, denote by τ_n the sequence of transactions occurring in the n^{th} row H_n of H. Let also τ_n^{D} denote the sequence consisting of only those transactions occuring in H_n whose corresponding status component is **dequeued**. The correctness of the processor can then be restated as follows.

Theorem. τ_∞^{D} *is an execution sequence.*

In view of Definition 3, this presents us with four proof obligations.

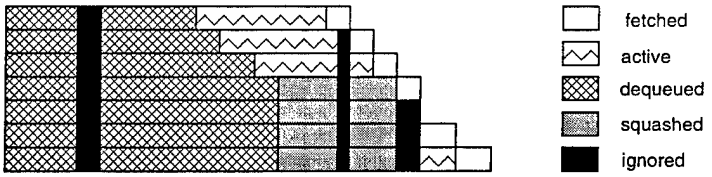


Fig. 4. Seven consecutive rows in the middle of a history table. The second depicts a cycle when only part of the fetched transactions is enqueued. The first misprediction is seen in the fourth row; transactions fetched at this cycle are ignored at the next, when also, due to the misprediction, the fetching unit was unable to output. ("Active" stands for statuses that are neither initial nor final and reflects the queue in the ordering unit.)

Proposition 1. *The sequence* τ_∞^{D} *is infinite.*

Proposition 2. *If* U *and* T *are two consecutive elements of* τ_∞^{D}*, then* $\mathsf{npc}(U) = \mathsf{addr}(T)$*. Also, the value or the* addr *field of the first transaction of* τ_∞^{D} *id* $\mathtt{pc}_{init}$*.*

Proposition 3. *Let* T *be a transaction in* τ_∞^{D}*. If* U *is the* r^{th} *register provider of* T *in* τ_∞^{D}*, then* $\mathsf{rOp}_r(T) = \mathsf{rRes}(U)$*, and if* T *does not have an* r^{th} *provider in* τ_∞^{D}*, then* $\mathsf{rSources}_r(T) = \mathtt{rf}_{init}(\mathsf{rSource}_r(T))$*.*

Proposition 4. *Let* T *be a transaction in* τ_∞^{D}*. If* U *is the store provider of* T *in* τ_∞^{D}*, then* $\mathsf{mOp}(T) = \mathsf{mRes}(U)$*, and if* T *does not have a store provider in* τ_∞^{D}*, then* $\mathsf{mOp}(T) = \mathtt{mem}_{init}(\mathsf{mSource}(T))$*.*

The proof of Proposition 1 uses the liveness conditions of components. The major results one needs to establish are the infinity of the sequences of fetched and enqueued transactions, and the absence of livelock, expressed as the statement that all locations in H_∞ are final. Proving the remaining three propositions involves a rather straightforward but tedious chasing around the history table.

7 Formal specification

Staying close to the *Hawk* specification style, we model processors and their components as state machines, which use sets of input wires, output wires, and states, each wire and each piece of state having a prescribed type. The machine is then defined by a function whose arguments are the values for input wires and states, and whose results are values for the output wires and states in the next clock cycle. Consequently, the machine acts as a signal transformer: for any given signals (infinite sequences) of inputs and initial values of states, it produces uniquely determined signals of outputs.

An axiomatic specification of a state machine could consist of a list of its input, output and state variables, an initial condition, an invariance condition, and a liveness condition. Without making these notions precise, we note that an

invariant is a propositional formula written in terms of input variables, output variables, state variables and primed state variables, and a liveness condition is a property of signals expressible by a suitable formula in temporal logic.

Again without going into technicalities, state machines can be composed by identifying each output wire of the constituent machines with some (zero or more) input wires. At the level of signals, which is how it is done in Hawk, composition amounts to writing a system of equations, each corresponding to a component machine.

The input, output and state variables of the three components of our processor can be read off from Fig. 1, which also tells how the wires are joined to give a specification of the processor as a composition of its components. The formulas for specifications of components are given below, after introducing notational conventions.

The values $\mathtt{pgm}$, $\mathtt{pc}_{init}$, $\mathtt{rf}_{init}$ and $\mathtt{mem}_{init}$ are constants.

We restrict the type **TransSeq** to "uniquely named" sequences: if two transactions in a sequence have names x and y, none of which is $\perp$, then $x \neq y$. The concatenation of sequences α and β is denoted $\alpha \# \beta$. A partial order on the set of transaction sequences is defined by $\alpha \preceq \beta$ if and only if $|\alpha| = |\beta|$ and $\alpha[i] \preceq \beta[i]$ for every i. A transaction is *mispredicting* if its spc and npc fields are not equal, and none is equal to $\perp$. A transaction is *decoded* if none of its fields opcode, rSources, rDest contains $\perp$. A transaction is *independent* if its mrSt and and rOps fields are maximal (the first is NONE and the second does not contain $\perp$). A transaction T *depends on* another transaction U if $\mathsf{rProv}_i(T) = \mathsf{name}(U)$ or $\mathsf{mrSt}(T) = \mathsf{name}(U)$. If T is a transaction in a transaction sequence α, then the *most recent store* of T in α is the last store in α that precedes T. Finally, if A is a transaction set and T is a transaction, then the *store chain* of T in A is the maximal sequence $\langle S_k, \ldots, S_1 \rangle$ with the properties $\mathsf{mrSt}(T) = \mathsf{name}(S_1)$ and $\mathsf{mrSt}(S_i) = \mathsf{name}(S_{i+1})$ for $1 \leq i < k$.

Transaction sets have the property that different elements of a set have distinct names; we use the type $\mathbf{TransSet} = (\mathbf{Name} - \{\perp\}) \to \mathbf{Trans}^\sharp$ to represent such sets. For A and B in **TransSet**, we denote by $A \cup B$ the union of A and B with A having the higher priority; that is, if A and B both have a transaction named x, then the transaction named x of $A \cup B$ is that of A. (This union operation is associative, but not commutative.) The notation $A \preceq B$ means by definition that $A(x) \preceq B(x)$ for every $x \in \mathbf{Name}$. Note that there is a canonical map $\mathbf{TransSeq} \to \mathbf{TransSet}$, so every transaction sequence can be regarded as a transaction set.

For $\mathtt{rf} \in \mathbf{Reg} \to \mathbf{Value}$, $\mathtt{mem} \in \mathbf{Addr} \to \mathbf{Value}$, $v \in \mathbf{Value}$, $r \in \mathbf{Reg}$ and $a \in \mathbf{Addr}$, the values of the updated register files and memories are denoted by $\mathtt{rf}[r \mapsto v]$ and $\mathtt{mem}[a \mapsto v]$. Note the role of $\perp$ in updating functions: if $\mathtt{rf}' = \mathtt{rf}[\perp \mapsto v]$, then $\mathtt{rf}'(r) = \perp$ for every r, but if $\mathtt{rf}' = \mathtt{rf}[r \mapsto \perp]$ then $\mathtt{rf}'(s) = \mathtt{rf}(s)$ for every $s \neq r$. Updating of a register file and memory by a transaction is defined by

$$\mathtt{rf} \cdot T = \begin{cases} \mathtt{rf}[\mathsf{rDest}(T) \mapsto \mathsf{rRes}(T)] & \text{if } \mathsf{rDest}(T) \in \mathbf{Reg} \\ \mathtt{rf} & \text{if } \mathsf{rDest}(T) = () \end{cases}$$

$$\mathtt{mem} \cdot T = \begin{cases} \mathtt{mem}[\mathsf{mDest}(T) \mapsto \mathsf{mRes}(T)] & \text{if } \mathsf{mDest}(T) \in \mathbf{Addr} \\ \mathtt{mem} & \text{if } \mathsf{mDest}(T) = () \end{cases}$$

The results $\mathtt{rf} \cdot \tau$ and $\mathtt{mem} \cdot \tau$ of updating rf and mem by a finite transaction sequence τ are then defined in a straightforward manner.

Fetching Unit

Let pc-rpc = pc if rpc = (); otherwise pc-rpc = rpc.

Fetch-Init. *The initial values of* pc *and* fetched *are* $\mathtt{pc}_{init}$ *and* $\langle\rangle$ *respectively.*

Fetch-Inv 1. $\mathsf{instr}(T) = \mathsf{pgm}(\mathsf{addr}(T))$, *for every transaction* T *occurring in* fetched.

Fetch-Inv 2 (Speculation). *If* $\mathtt{fetched} = \langle T_1, \ldots, T_k \rangle$, *then* $\mathsf{addr}(T_1) = \mathtt{pc\text{-}rpc}$, *and* $\mathsf{addr}(T_{i+1}) = \mathsf{spc}(T_i)$ *for every* $i \in \{1, \ldots, k-1\}$.

Fetch-Inv 3 (Next PC). $\mathtt{pc}' = \mathsf{spc}(T)$ *if* T *is the last transaction of* fetched, *and* $\mathtt{pc}' = \mathtt{pc\text{-}rpc}$ *if* $\mathtt{fetched} = \langle\rangle$.

Fetch-Inv 4 (Empty fields). *A field of a transaction in* fetched *has a value different from* $\perp$ *if and only if that field is* instr, spc *or* addr.

Fetch-Liv. *The formula* $\mathtt{rpc} \neq () \vee \mathtt{fetched} \neq \langle\rangle$ *is true infinitely often.*

Ordering Unit

Denote queue = mature $\#$ young.

Ord-Init. *The initial values of* xpc, rf, queue, flush, prepared *and* rpc *are* $\mathtt{pc}_{init}$, $\mathtt{rf}_{init}$, $\langle\rangle$, FALSE, $\langle\rangle$ *and* () *respectively.*

Ord-Inv 1 (Naming). *All transactions in* queue *have distinct names.*

Ord-Inv 2 (Queue). *Let* mature* *be the sequence obtained from* mature *by replacing every transaction in it with an equally named transaction of* computed, *if it exists. If* flush = TRUE *then* queue′ = prepared = $\langle\rangle$ *and* dequeued *is a prefix of* mature*. *If* flush = FALSE, *then there exists a prefix* enqueued *of* fetched *such that*

$$\mathtt{young} \,\#\, \mathtt{enqueued} \preceq \mathtt{prepared} \,\#\, \mathtt{young}',$$
$$\mathtt{mature}^* \,\#\, \mathtt{prepared} = \mathtt{dequeued} \,\#\, \mathtt{mature}'.$$

Ord-Inv 3 (Enqueueing). *If* T *is the first transaction of* enqueued, *then* $\mathsf{addr}(T) = \mathtt{xpc}$. *Finally, if* queue = $\langle\rangle$ *and* xpc = $\mathsf{addr}(T)$, *where* T *is the first transaction of* fetched, *then* enqueued $\neq \langle\rangle$.

Ord-Inv 4 (Preparation). *Let* T *be a transaction in* prepared. *Then*

1. *T is decoded,* $\mathsf{rRes}(T) = \bot$, *and* $T \in \mathtt{mature}'$.
2. $\langle \mathsf{rOp}_i(T), \mathsf{rProv}_i(T)\rangle = \langle \bot, \mathsf{name}(U)\rangle$ *if U is the i^{th} register provider of T in* queue$'$, *and* $\langle \mathsf{rOp}_i(T), \mathsf{rProv}_i(T)\rangle = \langle \mathtt{rf}'(\mathsf{rSource}_i(T)), \text{NONE}\rangle$ *if T does not have the i^{th} register provider in* queue$'$.
3. $\mathsf{mrSt}(T) = \mathsf{name}(S)$ *if S is the most recent store for T in* queue$'$, *and* $\mathsf{mrSt}(T) =$ NONE *if this most recent store does not exist. The value of* $\mathsf{mOp}(T)$ *is* $\bot$ *or* $()$, *depending on whether T is a load or not.*

Ord-Inv 5 (Dequeueing). *All transactions of* dequeued *are complete and none of them, except possibly the last one, is mispredicting.*

Ord-Inv 6 (Register File). $\mathtt{rf}' = \mathtt{rf} \cdot \mathsf{dequeued}$.

Ord-Inv 7 (Flush). $\mathtt{flush} =$ TRUE *if and only if the last transaction in* dequeued *is mispredicting.*

Ord-Inv 8 (Enabling a memory write). $\mathtt{writemem} =$ TRUE *if and only if the first transaction of* queue$'$ *is an incomplete store.*

Ord-Inv 9 (Requested PC).

$$\mathtt{rpc} = \begin{cases} \mathsf{npc}(D) & \text{if } \mathtt{flush} = \text{TRUE} \\ \mathsf{addr}(E) & \text{if } \mathtt{flush} = \text{FALSE and } |\mathsf{enqueued}| < |\mathtt{fetched}| \\ () & \text{otherwise} \end{cases},$$

where D is the last transaction of dequeued *and* $E = \mathtt{fetched}(|\mathsf{enqueued}| + 1)$.

Ord-Inv 10 (Expected PC).

$$\mathtt{xpc}' = \begin{cases} \mathtt{rpc} & \text{if } \mathtt{rpc} \neq () \\ \mathsf{spc}(T) & \text{if } \mathtt{rpc} = () \text{ and } \mathsf{enqueued} \neq \langle\rangle \\ \mathtt{xpc} & \text{otherwise} \end{cases},$$

where T is the last transaction of enqueued.

Ord-Liv. *If the first transaction of* queue *is complete, then eventually* $\mathsf{dequeued} \neq \langle\rangle$. *If* $\mathtt{mature} = \langle\rangle$ *and* $\mathsf{young} \neq \langle\rangle$, *then eventually* $\mathtt{prepared} \neq \langle\rangle$.

EXECUTION UNIT

Exec-Init. *The initial value of* mem *is* $\mathtt{mem}_{init}$, *and* $\emptyset$ *is the initial value of* executing, lingering *and* computed.

Exec-Inv 1 (Flushing). *If* $\mathtt{flush} =$ TRUE, *then* $\mathtt{executing}' = \mathtt{lingering}' = \emptyset$ *and* $\mathtt{mem}' = \mathtt{mem}$.

Exec-Inv 2 (Contents). *The sets* executing *and* lingering *are disjoint. If* $\mathtt{flush} =$ FALSE *then*

$$\mathtt{executing} \cup \mathsf{prepared} \cup \mathtt{lingering} \preceq \mathtt{executing}' \cup \mathtt{lingering}'. \quad (1)$$

If T is an element of the left-hand side of (1) and T' is the corresponding element of the right-hand side, we will say that T' is the *descendant* of T. Note that the only transactions of executing $\cup$ lingering without a descendant are members of lingering whose name occurs in a transaction of prepared.

Exec-Inv 3 (Lingering). *Assume* flush = FALSE. *Then all transactions in* lingering *are complete and no transaction in* executing *depends on any transactions of* lingering. *Also, a transaction belongs to* lingering′ *if and only if it either belongs to* computed, *or is a descendant of a transaction in* lingering.

If L is a load in executing $\cup$ prepared and ϕ is the store chain of L in this set, then

$$\mathsf{mOp}(L) \preceq (\mathtt{mem} \cdot \phi)(\mathsf{mSource}(L)) \qquad \text{(LC)}$$

is a condition that should be satisfied by the execution unit. Note that the value on the right-hand side is $\perp$ if $\mathsf{mDest}(S) = \perp$ for some S in ϕ. If $\mathsf{mDest}(S) \neq \perp$ for all S in ϕ, then the value on the right-hand side is either (1) $\mathsf{mRes}(S)$, where S is the last transaction in ϕ with $\mathsf{mDest}(S) = \mathsf{mSource}(L)$, or (2) $\mathtt{mem}(\mathsf{mSource}(L))$, if no such S exists.

Exec-Inv 4 (Load Correctness). *If L' is the descendant of a load L which satisfies the condition* (LC), *then L' satisfies* (LC) *too.*

Exec-Inv 5 (Forwarding). *If T' is the descendant of T, then* $\langle \mathsf{rProv}_i(T'), \mathsf{rOp}_i(T') \rangle = \langle \mathsf{rProv}_i(T), \mathsf{rOp}_i(T) \rangle$, *or* $\langle \mathsf{rProv}_i(T'), \mathsf{rOp}_i(T') \rangle = \langle \text{NONE}, \mathsf{rRes}(U) \rangle$, *where* $U \in$ executing $\cup$ lingering, $\mathsf{rProv}_i(T) = \mathsf{name}(U)$, *and* $\mathsf{rRes}(U) \neq \perp$.

Exec-Inv 6 (Memory). *1. If* $\mathtt{mem}' \neq \mathtt{mem}$, *then* writemem = TRUE *and* $\mathtt{mem}' = \mathtt{mem} \cdot S$, *where S is a complete store in* executing.
2. If computed *contains a store S, then* $\mathtt{mem}' = \mathtt{mem} \cdot S$ *and* writemem = TRUE.

Exec-Inv 7 (Most Recent Store). *If T' and U' are descendants of T and U, and if* $\mathsf{mrSt}(T) = \mathsf{name}(U)$, *then* $\mathsf{mrSt}(T') = \mathsf{name}(U')$ *unless* $U' \in$ computed *or T is a load with* $\mathsf{mOp}(T) \neq \perp$.

Exec-Liv. *Let T be an independent transaction in* executing. *If T is a store, assume also that* writemem = TRUE. *Then eventually* flush = TRUE *or* $\mathsf{name}(T)$ *occurs among names of transactions in* computed.

8 Conclusions

In an attempt to bring the power of verification closer to the complexity of commercial processors, we have specified a general microarchitectural design and proved its correctness. Our axiomatization can be satisfied by a family of microarchitectures; therefore, it retains a good deal of flexibility as the structure of the individual components is developed. Since each component is specified independent of other components, the implementation and proof of components can be carried out independently. Furthermore, our specifications and proof are

independent of many considerations that affect performance. For example, we do not need to set the number and latencies of subunits of our execution units, the width of instruction-carrying wires, the accuracy of branch prediction etc. Therefore, many design decisions based on simulation may be made without adversely affecting the global correctness proof. Note also that the wires present in our top-level specification are just what is necessary for interunit communication. The units are free to communicate through extra channels; for example, an extra wire allows implementation of a branch target buffer within the fetching unit.

Most of the advantages of our approach come as a consequence of using a severely minimized axiomatization. This approach is not quite common, probably because coming up with a reasonably complete set of invariants for an algorithm is generally difficult. Considerable skill is required to extract the axioms, but in a limited domain, such as that of hardware design, it could be feasible. We plan to explore the axiomatics for hardware components and develop a library of specifications and typical proofs.

We intend to construct various refinements of our component specifications and thus to show that our axiomatizations can be related to specific microarchitectures. We have already developed executable PentiumPro-like specifications in *Hawk* using the same structure described here (see [1]); we plan to prove the correctness of these executable models by checking their three units satisfy our axioms. Transactions, as we have demonstrated, are a useful microarchitectural abstraction, but they also come with a substantial overhead that should be eliminated in lower-level refinements. We plan to develop a methodology for shrinking the interfaces of our top-level specifications.

We expect that further research will confirm that reasoning around the history table is a promising proof technique, applicable to pipeline designs in general. Also left to further research is rewriting our axiomatics in a more stringent specification style, and mechanization of the proofs.

Acknowledgments. For their contributions to this research, we thank Mark Aagaard, Borislav Agapiev, Robert Jones, and John O'Leary of Intel Strategic CAD Labs; Tito Autrey, Nancy Day, Dick Kieburtz and Thomas Nordin of OGI; and Arvind of MIT.

The authors are supported by Intel Strategic CAD Labs and Air Force Material Command (F19628-93-C-0069). John Matthews receives support from a graduate research fellowship with the National Science Foundation.

References

[1] Hawk Web page: `http://www.cse.ogi.edu/PacSoft/Hawk/`.

[2] M. Aagaard, R. Jones, and C.-J. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In 35^{th} *Design Automation Conference (DAC '98)*, pages 538–541. Association for Computing Machinery, 1998.

[3] M. Aagaard and M. Leeser. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[4] Arvind and X. Shen. Design and verification of processors using term rewriting systems. *IEEE Micro*, 1999. to appear.

[5] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining symbolic model checking with uninterpreted functions for out-of-order processor verification. In [9], pages 369–386.

[6] J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–70. Springer-Verlag, 1994.

[7] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors with Hawk. In *Workshop on Formal Techniques for Hardware and Hardware-like Systems*, Marstrand, Sweden, June 1998.

[8] Á. P. Eiríksson. The formal design of 1M-gate ASICs. In [9], pages 49–63.

[9] G. Gopalakrishnan and P. Windley, editors. *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[10] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, 9(2):9–15, 1995.

[11] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In [13], pages 440–451.

[12] R. Hosabbettu, M. Srivas, and G. Gopalakrihnan. Decomposing the proof of correctness of pipelined microprocessors. In [13], pages 122–134.

[13] A. J. Hu and M. Y. Vardi, editors. *Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[14] R. B. Jones, J. U. Skakkebaek, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In [9], pages 2–17.

[15] J. Matthews, J. Launchbury, and B. Cook. Specifying microprocessors in Hawk. In *1998 International Conference on Computer Languages*, pages 90–101. IEEE Computer Society, 1998.

[16] K. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In [13], pages 110–121.

[17] J. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD K86. *IEEE Transactions on Computers*, 47(9):913–926, 1998.

[18] A. Pnueli and T. Arons. Verification of data-insensitive circuits: An in-order-retirement study. In [9], pages 351–568.

[19] J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In [13], pages 135–146.

[20] B. Shiver and B. Smith. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. IEEE Computer Society, 1998.

A Appendix: Correctness Proof

In Sect. 6 we gave a brief and incomplete description of the history table associated to a run of our processor model. A precise definition is given below in Subsect. A.3. In particular, we prove that the columns of the history table stabilize (Lemma 12), so that the sequence τ_∞ of limit values is defined. Recall that the sequence τ_∞^{D} is obtained by removing from τ_∞ all transactions whose corresponding status is not dequeued. We prove that this sequence is equal to the concatenation of all sequences of transactions dequeued by our processor in the run being considered (Lemma 16). Thus the correctness of the processor can indeed be expressed as in Theorem stated in Sect. 6. We repeat it here:

Theorem. *τ_∞^{D} is an execution sequence.*

We also repeat the four Propositions which, in view of Definition 3, imply the theorem.

Proposition 1. *The sequence τ_∞^{D} is infinite.*

Proposition 2. *If U and T are two consecutive elements of τ_∞^{D}, then $\mathsf{npc}(U) = \mathsf{addr}(T)$. Also, the value or the* addr *field of the first transaction of τ_∞^{D} id $\mathtt{pc}_{init}$.*

Proposition 3. *Let T be a transaction in τ_∞^{D}. If U is the r^{th} register provider of T in τ_∞^{D}, then $\mathsf{rOp}_r(T) = \mathsf{rRes}(U)$, and if T does not have an r^{th} provider in τ_∞^{D}, then $\mathsf{rSources}_r(T) = \mathtt{rf}_{init}(\mathsf{rSource}_r(T))$.*

Proposition 4. *Let T be a transaction in τ_∞^{D}. If U is the store provider of T in τ_∞^{D}, then $\mathsf{mOp}(T) = \mathsf{mRes}(U)$, and if T does not have a store provider in τ_∞^{D}, then $\mathsf{mOp}(T) = \mathtt{mem}_{init}(\mathsf{mSource}(T))$.*

The proofs of the propositions are given in Subsections A.5–A.8. The definition and some basic properties of the history table are given in Subsection A.3. The first two subsections contain notational preliminaries and key lemmas about the relationships among the processor's components.

A.1 Terminology

Regular and singular cycles. For a given run of the processor, the value of any state variable v at the cycle n ($n \geq 1$) will be denoted by v^n. Define n to be *regular* or *singular* depending on whether $\mathtt{flush}^n$ is FALSE or TRUE. Note that n is singular if and only if $\mathtt{dequeued}^n$ is non-empty and the last transaction in it is mispredicting (Ord-Inv 5). Note also that if n is singular, then $\mathtt{queue}^n$, $\mathtt{executing}^{n+1}$ and $\mathtt{executing}^{n+1}$ are empty, by Ord-Inv 2 and Exec-Inv 1 respectively. As a consequence, we have that two consecutive numbers cannot be both singular.

Locations. Let us use the term *location* for the four wires (fetched, prepared, computed, dequeued) and the four state elements (young, mature, executing, lingering) that serve as transaction holders in our processor's specification. In addition to these, we will also consider a few more defined "locations", some of which have previously been defined or just mentioned. First we have $\texttt{queue}^n = \texttt{mature}^n \,\#\, \texttt{young}^n$ and $\texttt{contents}^n = \texttt{executing}^n + \texttt{lingering}^n$, the full contents of the ordering and the execution units respectively. Then we have $\texttt{enqueued}^n$, a prefix of $\texttt{fetched}^n$, defined when n is regular and with properties given in Ord-Inv 2 and Ord-Inv 3. We define $\texttt{enqueued}^n = \langle\rangle$ when n is singular. Furthermore, we define $\texttt{ignored}^n$ by $\texttt{fetched}^n = \texttt{enqueued}^n \,\#\, \texttt{ignored}^n$ when n is regular, and $\texttt{ignored}^n = \langle\rangle$ when n is singular. $\texttt{ripe}^n$ is the transaction set consisting of complete transactions in $\texttt{mature}^n$. Finally, when n is regular, we define $\texttt{squashed}^n = \langle\rangle$, and when n is singular, we define $\texttt{squashed}^n$ to be the suffix of $\texttt{queue}^{n-1} \,\#\, \texttt{fetched}^{n-1}$ of length complementary to $|\texttt{dequeued}^n|$.

Note that the nine location names are used to name the nodes of the transaction diagram Γ in Fig. 3. If X and Y are two nodes of Γ we will write $X \leq Y$ if $X = Y$ or there exists a sequence of arcs in Γ leading from X to Y. There are no non-trivial cycles in Γ, so this is a partial order relation.

Ancestors and ordinals. A simple fundamental observation is that any transaction present in the processor at any cycle in any of the eight basic locations except fetched has a uniquely determined *immediate ancestor* among transactions present in the processor at the previous cycle. Note, however, that it is not realistic to assume that this relationship is "one-to-one". For example, in the model we are considering, each transaction in $\texttt{prepared}^n$ wire has a copy of itself saved in $\texttt{mature}^n$ and each transaction in $\texttt{executing}^n$ or $\texttt{computed}^n$ also has a copy of its ancestor waiting in $\texttt{mature}^n$. Choosing a unique "descendant" of a fetched instruction in all subsequent cycles is tantamount to the definition of the history table; see A.3.

Since the initial value X^1 is empty for every $X \neq \texttt{fetched}$, it follows that starting with any transaction T belonging to a location X^n one can define a sequence of transactions in which each is the immediate ancestor of the previous one and which terminates at a transaction T_0 belonging to $\texttt{fetched}^k$ for some $k \leq n$. This T_0 is a uniquely defined progenitor of T. The *ordinal* of T is defined to be the ordinal of T_0 in the sequence $\texttt{all-fetched} = \texttt{fetched}^1 \,\#\, \texttt{fetched}^2 \,\#\, \cdots$ of all fetched transactions.

It remains to give a precise definition of immediate ancestors. So suppose X is a basic location, $X \neq \texttt{fetched}$, and $T \in X^n$. We define the ancestor T' of T and its location Y^{n-1}. Consider first the possibilites executing, lingering and computed for X. If $n-1$ is regular, then T' and Y are found from the inequality

$$\texttt{executing}^{n-1} \cup \texttt{prepared}^{n-1} \cup \texttt{lingering}^{n-1} \preceq \texttt{contents}^n \qquad (2)$$

of Exec-Inv 2. If $n-1$ is singular, then $\texttt{executing}^n$, $\texttt{lingering}^n$ and $\texttt{computed}^n$ are empty, so there is nothing to define. Turning to the possibilities young,

mature, prepared and dequeued for X, we obtain the corresponding T' and Y easily from the relations

$$\texttt{young}^{n-1} \,\#\texttt{enqueued}^n \preceq \texttt{prepared}^n \,\#\texttt{young}^n, \tag{3}$$

$$\texttt{mature}^* \,\#\texttt{prepared}^n = \texttt{dequeued}^n \,\#\texttt{mature}^n. \tag{4}$$

of Ord-Inv 2, provided that n is regular. And if n is singular, then $\texttt{prepared}^n$, $\texttt{mature}^n$ and $\texttt{young}^n$ are empty (Ord-Inv 2) so there is nothing to do for them, while for $\texttt{dequeued}^n$ we have that it is a prefix of a sequence $\texttt{mature}^*$, where each member of $\texttt{mature}^*$ belongs to either $\texttt{mature}^{n-1}$ and $\texttt{computed}^{n-1}$.

Note that in all cases we have $T' \preceq T$.

A.2 Between processor units

From the informal specification of the ordering unit (Sect. 4) we expect that transactions in $\texttt{mature}^n$ should fall into four well-defined classes: for each T in $\texttt{mature}^n$, T is either complete and waiting for its turn to be dequeued, or there is a unique transaction associated (by name) with T in $\texttt{prepared}^n$, $\texttt{executing}^n$, or $\texttt{computed}^n$. Lemma 2 below confirms this basic relationship between the contents of the ordering and the execution units. Lemmas 3 and 4 state two important relationships between what comes in and what goes out. They refer to the execution unit and the ordring unit respectively, but neither can be derived from the axiomatics of a single unit.

First we need to extend our notation about transaction sets. Transaction sets are *disjoint* if their domains are disjoint as sets; we will write $A+B$ for $A \cup B$ in the case when we know A and B are disjoint. Define $A \setminus B$ to be the restriction of A on the set difference of the domains of A and B. Define A to be a *subset* of B if $A(x) = B(x)$ whenever $A(x) \neq ()$. We will write $A - B$ for $A \setminus B$ when we know that A is a subset of B.

Lemma 1. *If n and $n-1$ are regular, then*

$$\texttt{executing}^{n-1} \cup \texttt{prepared}^{n-1} \preceq \texttt{executing}^n + \texttt{computed}^n.$$

Proof. Since n is regular, Exec-Inv 2 implies

$$(\texttt{executing}^{n-1} \cup \texttt{prepared}^{n-1}) + (\texttt{lingering}^{n-1} \setminus \texttt{prepared}^{n-1}) \preceq \texttt{executing}^n + \texttt{lingering}^n.$$

Since $n-1$ is regular, Exec-Inv 3 implies

$$\texttt{lingering}^n = \texttt{computed}^n + (\texttt{lingering}^{n-1} \setminus \texttt{prepared}^{n-1}).$$

The lemma immediately follows from these relations. □

Lemma 2. *For every regular n, the sets $\texttt{ripe}^n$, $\texttt{computed}^n$, $\texttt{executing}^n$ and $\texttt{prepared}^n$ are disjoint, and*

$$\texttt{mature}^n \preceq \texttt{ripe}^n + \texttt{computed}^n + \texttt{executing}^n + \texttt{prepared}^n. \tag{5}$$

Moreover, the corresponding elements on the two sides have the same ordinal.

Proof. The proof is by induction. Since the initial values of all the sets involved are empty, the initial case is true. The induction step splits into two cases, depending on whether $n-1$ is regular or not.

Assume first $n-1$ is not regular. By Ord-Inv 2, we have $\texttt{mature}^{n-1} = \texttt{young}^{n-1} = \langle\rangle$ and then $\texttt{prepared}^n = \texttt{dequeued}^n \# \texttt{mature}^n$. This implies $\texttt{mature}^n = \texttt{prepared}^n$ because all transactions in $\texttt{dequeued}^n$ are complete and so cannot occur in $\texttt{prepared}^n$, which (by Ord-Inv 4) contains only incomplete transactions. It remains only to prove that the sets $\texttt{ripe}^n$, $\texttt{computed}^n$ and $\texttt{executing}^n$ are empty. For $\texttt{ripe}^n$ it is true because all elements of $\texttt{mature}^n$ are incomplete. The other two are subsets of $\texttt{contents}^n$ which is empty by Exec-Inv 1.

Assume now that $n-1$ is regular. By Ord-Inv 2, we have

$$\texttt{mature}^* + \texttt{prepared}^n = \texttt{dequeued}^n + \texttt{mature}^n, \tag{6}$$

where $\texttt{mature}^*$ is obtained by replacing every transaction in $\texttt{mature}^{n-1}$ with an equally named transaction of $\texttt{computed}^{n-1}$. By induction hypothesis, all names of $\texttt{computed}^{n-1}$ occur among names of $\texttt{mature}^{n-1}$, so we have

$$\texttt{mature}^* = \texttt{computed}^{n-1} + (\texttt{mature}^{n-1} \setminus \texttt{computed}^{n-1}). \tag{7}$$

Combining (6) and (7), and the induction hypothesis in the form

$$\texttt{mature}^{n-1} \setminus \texttt{computed}^{n-1} \preceq \texttt{ripe}^{n-1} + \texttt{executing}^{n-1} + \texttt{prepared}^{n-1},$$

we obtain

$$\begin{aligned}\texttt{dequeued}^n + \texttt{mature}^n \preceq\ & \texttt{computed}^{n-1} + \texttt{ripe}^{n-1} + \texttt{executing}^{n-1} \\ & + \texttt{prepared}^{n-1} + \texttt{prepared}^n.\end{aligned} \tag{8}$$

Observe now that $\texttt{ripe}^{n-1} + \texttt{computed}^{n-1}$ is the set of complete transactions in $\texttt{mature}^*$; this follows from (7), the fact that all transactions in $\texttt{computed}^{n-1}$ are complete, and the induction hypothesis implying that the complete transactions in $\texttt{mature}^{n-1} \setminus \texttt{computed}^n$ are precisely those of $\texttt{ripe}^{n-1}$. Since no transaction of $\texttt{prepared}^n$ is complete (Ord-Inv 4) and all transactions of $\texttt{dequeued}^n$ are complete (Ord-Inv 5), it follows from (6) that the same set of complete transactions of $\texttt{mature}^*$ can also be written as $\texttt{dequeued}^n + \texttt{ripe}^n$. Thus, (8) rewrites into

$$\begin{aligned}\texttt{dequeued}^n + \texttt{mature}^n \preceq\ & \texttt{dequeued}^n + \texttt{ripe}^n + \texttt{executing}^{n-1} \\ & + \texttt{prepared}^{n-1} + \texttt{prepared}^n,\end{aligned}$$

and the desired result follows immediately from Lemma 1.

It remains to go back and check that the ordinals are the same for any two correspondind members of the two sides of any equality and inequality that was used in the proof. This is done by a straighforward inspection. □

Lemma 3. *If n and $n-1$ are regular, then*

$$\texttt{executing}^{n-1} + \texttt{prepared}^{n-1} \preceq \texttt{executing}^n + \texttt{computed}^n.$$

Proof. This is a strengthening of Lemma 1; that $\texttt{prepared}^{n-1}$ and $exec^{n-1}$ are disjoint is a part of Lemma 2. □

Lemma 4. *If n and $n-1$ are regular, then*

$$\texttt{computed}^n + \texttt{ripe}^n = \texttt{dequeued}^{n+1} + \texttt{ripe}^{n+1} \tag{9}$$

and all transactions of this set belong to $\texttt{lingering}^{n+1}$.

Proof. The equation is proved in the course of proving Lemma 2. As in the proof of Lemma 1, we have

$$\texttt{lingering}^n = \texttt{computed}^n + (\texttt{lingering}^{n-1} \setminus \texttt{prepared}^{n-1}),$$

so all we need to prove is that $\texttt{ripe}^n$ is a subset of $\texttt{lingering}^{n-1} \setminus \texttt{prepared}^{n-1}$. Arguing by induction, the problem reduces to showing that the sets $\texttt{ripe}^n$ and $\texttt{prepared}^{n-1}$ are disjoint. Indeed, by Exec-Inv 2 and Exec-Inv 3, every transaction in $\texttt{prepared}^{n-1}$ has a descendant in $\texttt{executing}^n$ or $\texttt{computed}^n$, and by Lemma 2, these two sets are disjoint from $\texttt{ripe}^n$. □

A.3 Definition of the history table

The top row of Fig. 3 depicts all possible paths through selected processor locations that a normally completed transaction can have, form fetching through retiring. A transition from X to Y in most cases should be interpreted as "it is possible that a transaction in X^n has a corresponding transaction in Y^{n+1}". The diagram also suggests that all transactions in X^n should have a corresponding transaction in some Y^{n+1} for some Y, the target node of an arc coming from X. "Corresponding" here means having the same ordinal, *i.e.*, being related to the same fetched instruction. Our goal is to define the history of execution of any fetched instruction, so we would like to define "transitions" $(T, X) \rightsquigarrow (T', Y)$ with (T', Y) uniquely determined by (T, X). When more than one such transition is possible, we select the right one according to the values of "control variables" (`flush` in our case).

Transaction flow. The subgraphs of Γ defined in Figs. 5–7 represent the transaction flow between cycles n and $n+1$, depending on whether these numbers are regular or singular. The following lemma states this in precise terms.

Lemma 5. *Let $n \geq 2$ and*

$$\Gamma_n = \begin{cases} \Gamma_{rr} & \text{if both } n-1 \text{ and } n \text{ are regular} \\ \Gamma_{rs} & \text{if } n \text{ is singular} \\ \Gamma_{sr} & \text{if } n-1 \text{ is singular} \end{cases}$$

and let

$$In_n = \{(T, X) \mid T \in X^{n-1} \text{ and } X \text{ is the source of an arrow of } \Gamma_n\},$$
$$Out_n = \{(T', Y) \mid T' \in Y^n \text{ and } Y \text{ is the target of an arrow of } \Gamma_n\}.$$

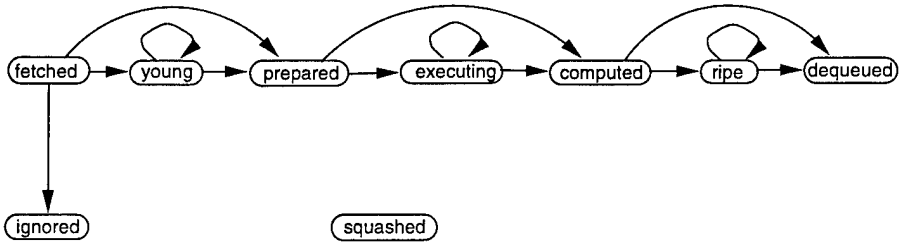


Fig. 5. Γ_{rr}.

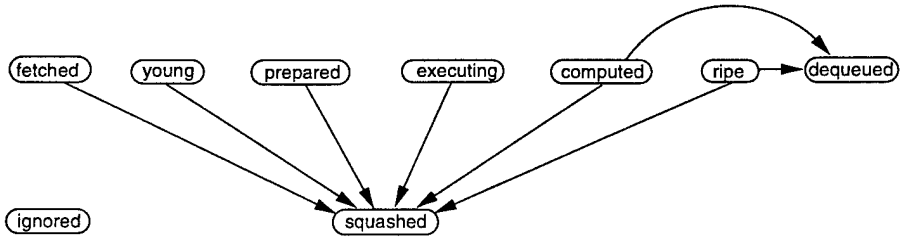


Fig. 6. Γ_{rs}.

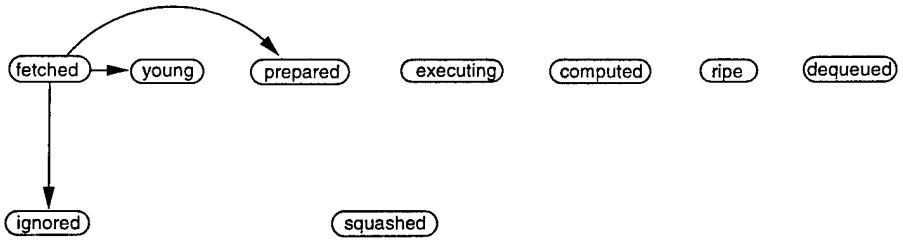


Fig. 7. Γ_{sr}.

Then the relation "have the same ordinal" defines a bijection $\delta_n: In_n \to Out_n$. Moreover, if $\delta_n(T,X) = (T',Y)$, then X and Y are joined by an arrow in Γ_n, and, in the cases $\Gamma_n = \Gamma_{rr}$ and $\Gamma_n = \Gamma_{sr}$, $T \preceq T'$.

Proof. We claim that if $n+1$ is regular, then

$$\texttt{young}^n \,\#\, \texttt{fetched}^n \preceq \texttt{prepared}^{n+1} \,\#\, \texttt{young}^{n+1} \,\#\, \texttt{ignored}^{n+1}, \qquad (10)$$

and if both n and $n+1$ are regular, then

$$\texttt{prepared}^n + \texttt{executing}^n \preceq \texttt{executing}^{n+1} + \texttt{computed}^{n+1}, \qquad (11)$$

$$\texttt{computed}^n + \texttt{ripe}^n = \texttt{dequeued}^{n+1} + \texttt{ripe}^{n+1}. \qquad (12)$$

Indeed, (10) follows from (3) and $\texttt{fetched}^n = \texttt{enqueued}^{n+1} \,\#\, \texttt{ignored}^{n+1}$, and (11) and (12) follow from Lemma 3 and Lemma 4 respectively. The case of the lemma when $\Gamma_n = \Gamma_{rr}$ immediately follows from these relations. Since $\texttt{young}^n = \langle\rangle$ when n is singular, the case $\Gamma_n = \Gamma_{rs}$ follows from (10) alone. Finally, in the case when $\Gamma_n = \Gamma_{sr}$ we have that $\texttt{squashed}^{n+1}$ is the suffix of $\texttt{mature}^n \,\#\, \texttt{young}^n \,\#\, \texttt{fetched}^n$ of length complementary to the length of $\texttt{dequeued}^{n+1}$ and that $\texttt{dequeued}^{n+1}$ is a prefix of $\texttt{mature}^*$, the sequence obtained by updating $\texttt{mature}^n$ with transactions of $\texttt{computed}^n$. The lemma now easily follows from Lemma 2. □

History table. Recall the definition of ordinals of transactions. In particular, transactions in the sequence $\texttt{all-fetched} = \texttt{fetched}^1 \,\#\, \texttt{fetched}^2 \,\#\, \cdots$ have distinct ordinals. For every $i \geq 1$, we define the *nascency rank* $\texttt{nr}(i)$ to be the number n such that $\texttt{fetched}^n$ contains a transaction with ordinal i. (If $\texttt{all-fetched}$ is finite, then $\texttt{nr}(i)$ would be defined only for $i \leq |\texttt{all-fetched}|$, but we will prove that $\texttt{all-fetched}$ is infinite, so $\texttt{nr}$ is defined for every positive integer.)

Definition 4. *For a given run of the processor and every n and i such that $n \geq \texttt{nr}(i)$ define H_n^i inductively as follows:*

1. *If $n = \texttt{nr}(i)$, then $H_n^i = (T, \texttt{fetched})$, where T is the transaction in $\texttt{fetched}^n$ whose ordinal is i.*
2. *If $H_{n-1}^i = (T,X)$ and X is a final location, then $H_n^i = H_{n-1}^i$.*
3. *If $H_{n-1}^i = (T,X)$ and X is not final, then $H_n^i = \delta_{n-1}(H_{n-1}^i)$.*

The history table *H is the table whose element belonging to the n^{th} row and the i^{th} column is H_n^i.*

The sequence of elements occuring in the n^{th} row of H will be denoted by H_n. The transaction and the status components of H_n^i will be denoted T_n^i and X_n^i respectively. The sequence of transaction components of H_n will be denoted τ_n and the sequence of the status components of H_n will be denoted ξ_n.

Lemma 6. *The definition of the history table is correct.*

Proof. The only thing that needs to be checked is that if $H^i_{n-1} = (T, X)$ and X is not final, then (T, X) belongs to In_n, the domain of δ_n. If $X = \texttt{fetched}$, then this is obvious. Otherwise, $(T, X) = \delta_{n-1}(T', X')$, so $(T, X) \in Out_{n-1}$. Thus, X is a target of an arrow in Γ_{n-1} and (by inspection of the eight possibilities for Γ_{n-1} and Γ_n) it follows that X is a source of an arrow of Γ_n, finishing the proof. □

A.4 Basic properties of the history table

Lemma 7. *If H^i_n is defined then*

1. $X^i_n \leq X^i_{n+1}$;

2. $T^i_n \preceq T^i_{n+1}$, *provided* $X^i_{n+1} \neq \texttt{squashed}$.

Proof. The proof follows immediately from Definition 4 and Lemma 5. □

Lemma 8. *All elements of Out_n occur in H_n.* □

Proof. The proof is obtained by strengthening the last argument in the proof of Lemma 6 by using bijectivity of δ_n. □

Note that the statuses related to the execution unit (`prepared`, `executing`, `computed`) do not occur in H_n when n is singular, so that the descendancy relation of Sect. 7 is not exactly reflected in the history table. In transitions between regular cycles, however, the descendancy in the execution unit can be seen in the table, as stated in the following lemma, easily derived from definitions.

Lemma 9. *If n and $n+1$ are both regular and X^i_n is* `prepared` *or* `executing`, *then T^i_{n+1} is the descendant of T^i_n (in the sense of Sect. 7).* □

Let **Act** denote the set consisting of the five nodes of Γ that are neither initial nor final. Let $\texttt{active}^n$ be the sequence obtained from τ_n by removing all transactions whose corresponding location is not in **Act**.

Lemma 10. *For every n, $\texttt{queue}^n \preceq \texttt{active}^n$. Moreover, if n is regular, then $\texttt{active}^n$ is the sequence obtained by replacing every transaction in $\texttt{mature}^n$ with an equally named transaction in the set $\texttt{computed}^n + \texttt{executing}^n$.*

Proof. Suppose first n is singular. Since $\texttt{queue}^n = \langle\rangle$, we need to show that $\texttt{active}^n = \langle\rangle$ too. Indeed, all elements of Out_n are of the form (T, X), where X is either `dequeued` or `squashed` (Fig. 6), and by Definition 4, the status of all elements in the n^{th} row of H is final.

Suppose now n is regular. By Lemma 8, if $X \in \textbf{Act}$ and $T \in X^n$, then (T, X) occurs in H_n. Then, by Lemma 2, there exists a bijection $T \mapsto T'$ between elements of $\texttt{active}^n$ and $\texttt{queue}^n$ such that $T' \preceq T$. All that remains to prove is that the elements of $\texttt{queue}^n = \texttt{mature}^n \,\#\, \texttt{young}^n$ have increasing ordinals, and that follows easily from the definitions of ancestors and ordinals. □

Now we can derive an often used lemma that guarantees existence of regular number intervals.

Lemma 11. *If* $X_n^i <$ dequeued *and* $\mathrm{nr}(i) < m < n$, *then* m *is regular.*

Proof. Since $m > nr(i)$, we have $X_m^i >$ fetched. Since $m < n$, we have $X_m^i <$ dequeued (Lemma 7). Thus, $X_m^i \in$ **Act** and so $\mathtt{active}^n \neq \langle\rangle$. Since $\mathrm{queue}^n = \langle\rangle$ when n is singular (Ord-Inv 2), the result follows from Lemma 10. □

The following is a stabilization lemma for columns of the history table.

Lemma 12. *For every* i, *the sequence* H_n^i *is eventually constant.*

Proof. Let $H_n^i = (T_n, X_n)$. By Lemma 7 $X_n \leq X_{n+1}$ in Γ. Since Γ is finite and the only cycles in it are loops at nodes, it follows that the sequence X_n stabilizes at n_0, say. Let X be its limit value. If X is final, then, again by Definition 4, H_n^i stabilizes as well. The remaining possibility is that X is young, executing or ripe. By Lemma 11, all numbers greater than n_0 are regular. By Lemma 5, we then have $T_n \preceq T_{n+1}$ for all $n > n_0$. By definition of progress ordering, all strictly increasing chains of transactions are finite, so the sequence T_n is eventually constant. □

The cycle at which the sequence H_n^i assumes its stable value will be denoted $\mathtt{sr}(i)$, the *stabilization rank.* The *limit row* H_∞ is the sequence of stable values: $H_\infty^i = \lim_n H_n^i$. The sequence of transactions and the sequence of statuses occurring in H_∞ will be denoted τ_∞ and ξ_∞.

Lemma 13. *The sequence* $\mathrm{dequeued}^{n+1}$ *is a prefix of* active^n.

Proof. By Ord-Inv 2, $\mathrm{dequeued}^{n+1}$ is a prefix of $\mathtt{mature}^*$, the sequence obtained by replacing transactions in $\mathtt{mature}^n$ with equally named transactions of $\mathrm{computed}^n$. When n is singular, $\mathrm{dequeued}^{n+1}$ is empty because $\mathtt{mature}^n$ is empty. When n is regular, the corollary follows from Lemma 10. □

Let τ_n^+ be the sequence obtained from τ_n by deleting all members whose corresponding status is ignored. Let also τ_n^{DS} be the sequence obtained from τ_n by keeping only its members whose status is dequeued or squashed.

Lemma 14. $\tau_n^+ = \tau_n^{\mathsf{DS}} \,\#\mathrm{active}^n \,\#\mathrm{fetched}^n$.

Proof. The proof is by induction. Assume τ_n^+ has the given form. By Definition 4, $\mathrm{fetched}^{n+1}$ is a suffix of τ_{n+1}^+. The prefix τ_n^{DS} remains intact in τ_{n+1}^+, by the same definition.

By Lemma 13, $\mathrm{dequeued}^{n+1}$ occurs as a prefix in active^n and so, by Definition 4, it will occur at the corresponding places in τ_{n+1}. Therefore, the sequence $\tau_n^{\mathsf{DS}} \,\#\mathrm{dequeued}^{n+1}$ is a prefix of τ_{n+1}^+. Now, if $n+1$ is regular, then, for each T_n^i of active^n which does not occur in the prefix $\mathrm{dequeued}^{n+1}$, we have $X_{n+1}^i \in$ **Act** (diagram Γ_{rr} or Γ_{sr}, though in the latter case there are no such elements T_n^i).

Also, if T_n^i is in $\texttt{fetched}^n$, then $X_{n+1}^i \in \mathbf{Act}$ or $X_{n+1}^i = \texttt{ignored}$. This finishes the proof if $n+1$ is regular. If $n+1$ is singular, then for every T_n^i in $\texttt{active}^n \,\#\texttt{fetched}^n$ that does not belong to the prefix $\texttt{dequeued}^{n+1}$, one has $X_{n+1}^i = \texttt{squashed}$ (diagram Γ_{rs}). □

As an immediate consequence of this lemma and its proof, we obtain the following.

Lemma 15. *For every* $n > 1$, $\tau_n^{\mathsf{DS}} = \tau_{n-1}^{\mathsf{DS}} \,\#\texttt{dequeued}^n \,\#\texttt{squashed}^n$. *If* n *is singular, then* $\tau_n^{\mathsf{DS}} = \tau_n^+$. □

Lemma 16. $\tau_n^{\mathsf{D}} = \texttt{dequeued}^1 \,\# \cdots \#\texttt{dequeued}^n$ *and* τ_n^{D} *is a prefix of* τ_∞^{D}.

Proof. By induction, using Lemma 15. □

Let ξ_n^+ be the sequence obtained from ξ_n by deleting all members equal to ignored.

Lemma 17. *For every* n, *the sequence* ξ_n^+ *regarded as a string, belongs to the set defined by the regular expression*

$$\{\texttt{dequeued}, \texttt{squashed}\}^*\{\texttt{executing}, \texttt{computed}, \texttt{ripe}\}^*\{\texttt{prepared}\}^*\{\texttt{young}\}^*\{\texttt{fetched}\}^*.$$

Moreover, if n *is singular, then the regular expression can be restricted to*

$$\{\texttt{dequeued}, \texttt{squashed}\}^*\{\texttt{fetched}\}^*.$$

Proof. The lemma follows from Lemma 15 and a simple observation that the sequence $\texttt{prepared}^n \,\#\texttt{ignored}^n$ is a suffix of $\texttt{queue}^n$ that occurs also as a suffix in $\texttt{active}^n$ (see Lemma 10). □

All transactions in $\texttt{fetched}^n$ have maximal values in their fields instr, addr and spc, and the field name is maximal in transactions of $\texttt{young}^n$. In $\texttt{prepared}^n$, all transactions have maximal values in their fields opcode, rSources and rDest. In $\texttt{computed}^n$ transactions are complete and so have maximal values in all fields. Combining these observations with Lemma 7, we obtain the following lemma, often used without being explicitly mentioned.

Lemma 18. *Fix* i, *let* $p, q \geq \mathrm{nr}(i)$ *and denote* $H_p^i = (T_p, X_p)$, $H_q^i = (T_q, X_q)$.

1. $\mathsf{field}(T_p) = \mathsf{field}(T_q) \neq \perp$ *for any* $\mathsf{field} \in \{\mathsf{instr}, \mathsf{addr}, \mathsf{spc}\}$
2. *If* $p, q > \mathrm{nr}(i)$, *then* $\mathsf{name}(T_p) = \mathsf{name}(T_q) \neq \perp$.
3. *If* $\texttt{prepared} \leq X_p, X_q \leq \texttt{dequeued}$, *then* $\mathsf{field}(T_p) = \mathsf{field}(T_q) \neq \perp$ *for any* $\mathsf{field} \in \{\mathsf{opcode}, \mathsf{rSources}, \mathsf{rDest}\}$
4. *If* $\texttt{computed} \leq X_p, X_q \leq \texttt{dequeued}$, *then* $T_p = T_q$.

□

Lemma 19. *If $i \leq j$, then* $\mathtt{nr}(i) \leq \mathtt{nr}(j)$. *If $i \leq j$ and $X^j_\infty \neq$* ignored, *then* $\mathtt{sr}(i) \leq \mathtt{sr}(j)$.

Proof. The first statement is an immediate consequence of the definition of nr. For the second statement, if $X^i_\infty =$ ignored, then $\mathtt{sr}(i) = \mathtt{nr}(i) + 1$ and $\mathtt{sr}(i) < \mathtt{sr}(j)$ easily follows. The interesting case, when neither of X^i_∞, X^j_∞ is ignored, follows from Lemma 16. □

Lemma 20. *Suppose $i < j$ and $X^i_n, X^j_n \in$* **Act**. *Then $X^j_\infty =$* dequeued *implies that $X^i_\infty =$* dequeued.

Proof. Suppose the lemma is not true. By Lemma 17, we must have $X^i_n =$ squashed. Consequently, $\mathtt{sr}(i)$ is singular. By Lemma 19, $n < \mathtt{sr}(i) \leq \mathtt{sr}(j)$. Lemma 11 discards all but the possibility $\mathtt{sr}(i) = \mathtt{sr}(j)$. This, however, contradicts the definition of $\mathtt{squashed}^n$ (dequeued transactions precede transactions squashed at the same cycle). □

A.5 Proof of Proposition 1

Lemma 21. $\Box\Diamond$ fetched $\neq \langle\rangle$.

Proof. Assume the contrary: $\Diamond\Box$ fetched $= \langle\rangle$. It follows then from (10) that $\Diamond\Box$ prepared $= \langle\rangle$. Then (11) implies $\Diamond\Box$ computed $= \langle\rangle$, and then (12) implies $\Diamond\Box$ dequeued $= \langle\rangle$. Now from Ord-Inv 9 we deduce $\Diamond\Box$ rpc $= ()$ and reach a contradiction with Fetch-Liv. □

Lemma 22. $\Box\Diamond$ queue $\neq \langle\rangle$.

Proof. Assume, on the contrary, that $\Diamond\Box$ queue $= \langle\rangle$. Then, by Ord-Inv 2, $\Diamond\Box$ dequeued $= \langle\rangle$ and then, by Ord-Inv 7, $\Diamond\Box$ flush $=$ FALSE. Also by Ord-Inv 2, $\Diamond\Box$ enqueued $= \langle\rangle$. By Lemma 21, there exists i such that $\mathtt{fetched}^i \neq \langle\rangle$, while $\mathtt{queue}^k = \mathtt{enqueued}^k = \mathtt{dequeued}^k = \langle\rangle$ and $\mathtt{flush}^k =$ FALSE for all $k \geq i$. Let $x = \mathtt{rpc}^{i+1}$. By Ord-Inv 9, $x \neq ()$, while $\mathtt{rpc}^{i+1} = ()$ for all $k > i + 1$. By Ord-Inv 10, $\mathtt{xpc}^k = x$ for all $k > i$. Now let j be the smallest number greater than i such that $\mathtt{fetched}^j \neq \langle\rangle$; it exists by Lemma 21. We have $\mathtt{pc\text{-}rpc}^{i+1} = x$ and, by repeated application of Fetch-Inv 3, $\mathtt{pc\text{-}rpc}^j = x$ as well. If T is the first transaction of $\mathtt{fetched}^j$, then $\mathtt{addr}(T) = x$ (Fetch-Inv 2), and then Ord-Inv 3 implies that $\mathtt{enqueued}^{j+1} \neq \langle\rangle$, which is a contradiction. □

Lemma 23. *All locations occurring in the entries of H_∞ are final.*

Proof. Since each of fetched, prepared and computed can occur at most once in any given column of the history table, none of them can occur in ξ_∞. We need to eliminate the possibility of occurrences of young, executing and ripe. Assuming the contrary, let k be the smallest integer such that $X^k_\infty = X$ is one of these three and let $m = \mathtt{sr}(k)$. Then $H^k_\infty = (T, X)$ for some T and, by Lemma 10, $\mathtt{queue}^n$ begins with a transaction T_n such that $T_n \preceq T$, for all $n > m$.

Case 1: $X = \texttt{young}$. Now we have $\texttt{young}^m \neq \langle\rangle$ and so $\texttt{mature}^n = \langle\rangle$ for all $n \geq m$. This implies $\texttt{prepared}^n = \langle\rangle$ for all $n \geq m$, directly contradicting Ord-Liv.
Case 2: $X = \texttt{ripe}$. We have $\texttt{dequeued}^n = \langle\rangle$ for all $n \geq m$. By Lemma 2, T_m equals T and so is complete. This again contradicts Ord-Liv.
Case 3: $X = \texttt{executing}$. Note first that, by Ord-Inv 8, $\texttt{writemem}^m = \text{TRUE}$ if T is a store; indeed, T_n is a store and is not complete since that would imply $X = \texttt{ripe}$. Secondly, by Lemma 11, all numbers greater than m are regular. Thirdly, since $\texttt{executing}^n$ and $\texttt{computed}^n$ are disjoint, $\mathsf{name}(T)$ does not occur in $\texttt{computed}^n$ for any $n \geq m$. These three facts, combined with Exec-Liv imply that T is not independent. Thus, we have (1) $\mathsf{rOp}_i(T) = \perp$ for some i, or (2) $\mathsf{mrSt}(T) \neq \text{NONE}$. Since $m = \mathtt{sr}(k)$ and $X^k_m = \texttt{computed}$, it follows that $X^k_{m-1} = \texttt{prepared}$.

If (1) holds, then by Ord-Inv 4, there exists U in $\texttt{queue}^{m-1}$ such that $\mathsf{rProv}_i(T^k_{m-1}) = \mathsf{name}(U)$. If (2) holds, then, again by Ord-Inv 4, there exists U in $\texttt{queue}^n$ such that $\mathsf{mrSt}(T^k_{m-1}) = \mathsf{name}(U)$. In both cases we have that T^k_{m-1} depends on T^j_n for some $j < k$. Since $j < k$ and all numbers greater than n are regular, we have $X^j_\infty = \texttt{dequeued}$ and so there exists n such that T^j_n is in $\texttt{computed}^n$. Then T^j_n also occurs in $\texttt{lingering}^n$ (Exec-Inv 3). By Lemma 18, $\mathsf{name}(T^j_n) = \mathsf{name}(T^j_{m-1})$, so $T^k_n = T$ depends on T^j_n, contradicting the axiom that a transaction in $\texttt{executing}^n$ cannot depend on any transaction in $\texttt{lingering}^n$ (Exec-Inv 3). □

Proof of Proposition 1. If i is the ordinal of a transaction in $\texttt{queue}^n$, Lemma 23 implies that X^i_∞ is either dequeued or squashed. It follows then from Lemma 22 that ξ_∞ contains infinitely many entries equal to dequeued or squashed. In other words, the sequence $\tau^{\mathsf{DS}}_\infty$ is infinite. By Lemma 15, this sequence is the concatenation of all sequences $\texttt{dequeued}^n \# \texttt{squashed}^n$. Since $\texttt{dequeued}^n \neq \langle\rangle$ whenever $\texttt{squashed}^n \neq \langle\rangle$ (by definition of squashed and Ord-Inv 7), it follows that $\texttt{dequeued}^n \neq \langle\rangle$ for infinitely many values for n, and therefore ξ^{D}_∞ is infinite.

A.6 Proof of Proposition 2

Let T^i_∞ and T^j_∞ be two consecutive elements of τ^{D}_∞. We need to prove that $\mathsf{npc}(T^i_\infty) = \mathsf{addr}(T^j_\infty)$. Let $m = \mathtt{sr}(i)$, $n = \mathtt{sr}(j)$, $m' = \mathtt{nr}(i)$, and $n' = \mathtt{nr}(j)$; by Lemma 19, we have $m \leq n$ and $m' \leq n'$.

First we show that every p between m and n (if it exists) is regular. Assume the contrary: there exists a singular p such that $m < p < n$. Then $\texttt{dequeued}^p \neq \langle\rangle$, by Ord-Inv 5. Thus, there exists l such that $\mathtt{sr}(l) = p$ and $X^l_\infty = \texttt{dequeued}$. By Lemma 19, it follows that $i < l < j$, contradicting the assumption that T^i_∞ and T^j_∞ are consecutive in τ^{D}_∞.

Assume first that T^i_∞ is not mispredicting; the other case will be considered separately. Since now $\mathsf{npc}(T^i_\infty) = \mathsf{spc}(T^i_\infty)$, all we need to show is $\mathsf{spc}(T^i_\infty) = \mathsf{addr}(T^j_\infty)$. If $m' = n'$ then $T^i_{m'}$ and $T^j_{m'}$ are members of $\texttt{fetched}^{m'}$ and both belong to $\texttt{enqueued}^{m'+1}$. Using Lemma 20, we deduce that these two transactions

must be consecutive in $\texttt{fetched}^{m'}$.Therefore, $\mathsf{spc}(T^i_{m'}) = \mathsf{addr}(T^j_{m'})$, by Fetch-Inv 2. Now assume $n' > m'$. Then $T^i_{m'+1}$ is the last element of $\texttt{enqueued}^{m'+1}$, $T^j_{n'+1}$ is the first element of $\texttt{enqueued}^{n'+1}$, and $\texttt{enqueued}^r = \langle\rangle$ for all r between $m'+1$ and $n'+1$ (if there are any such r). We claim that all r between m' and n' are regular. We know that all numbers between m' and m are regular (Lemma 11), and that all numbers between m and n are regular (proved above). Thus, the claim fails only if m is singular and $n' > m$. Then T^i_m would be the last transaction in $\texttt{dequeued}^m$ (because $n > m$ now and $\texttt{dequeued}^r = \langle\rangle$ for all r between m and n), contradicting the assumption that T^i_∞ is not mispredicting.

We can conclude that $\texttt{xpc}^{m'+1} = \mathsf{spc}(T^i_{m'})$ from Ord-Inv 10, that $\texttt{xpc}^{m'+1} = \cdots = \texttt{xpc}^{n'}$ (also from Ord-Inv 10), and that $\texttt{xpc}^{n'} = \mathsf{addr}(T^j_{n'+1})$ from Ord-Inv 3. This finishes the proof in the case when T^i_∞ is not mispredicting.

Assume finally that T^i_∞ is mispredicting. It follows from Ord-Inv 5 that m is singular and also that $\texttt{xpc}^{m+1} = \mathsf{npc}(T^i_m)$ (Ord-Inv 9 and Ord-Inv 10). It follows also that $n' \geq m$; otherwise T^j_m would exist and would be in $\texttt{active}^m$, which is absurd because this sequence must be empty since m is singular.

It follows that $T^j_{n'+1}$ is the first element of $\texttt{enqueued}^{n'+1}$ and that $\texttt{enqueued}^r = \langle\rangle$ for every r between m and $n'+1$. We already know that the numbers between m and $n'+1$ are all regular, and it follows from the Ord-Inv axioms, similarly as in the previous case, that $\texttt{xpc}^{m+1} = \cdots = \texttt{xpc}^{n'+1} = \mathsf{addr}(T^j_{n'+1})$.

We also need to prove that $\mathsf{addr}(T^1_\infty) = \texttt{pc}_{init}$. We do have $\texttt{pc}^1 = \texttt{pc}_{init}$ by Fetch-Init. Let $n = \texttt{nr}(1)$. By Fetch-Inv 2, $\mathsf{addr}(T^1_n) = \texttt{pc}^{n-1}$. Since $\mathsf{addr}(T^1_\infty) = \mathsf{addr}(T^1_n)$ (Lemma 18), it suffices to check that $\texttt{pc}^{n-1} = \texttt{pc}^1$. In view of Fetch-Inv 3, this reduces to proving $\texttt{rpc}^m = ()$ for $1 \leq m < n-1$. The last claim is a consequence of Ord-Inv 9 and simple facts $\texttt{flush}^m = \text{FALSE}$ and $\texttt{queue}^m = \langle\rangle$ for all $m < n-2$.

A.7 Proof of Proposition 3

Lemma 24. *For every* $\texttt{reg} \in \mathbf{Reg}$ *and* $n \geq 1$,

$$\texttt{rf}^n(\texttt{reg}) = \begin{cases} \mathsf{rRes}(T) & \text{if } T \text{ is the last element in } \tau^{\mathsf{D}}_n \text{ such that } \mathsf{rDest}(T) = \texttt{reg} \\ \texttt{rf}_{init}(\texttt{reg}) & \text{if such } T \text{ does not exist} \end{cases}.$$

Proof. The proof follows from Lemma 16 and Ord-Inv 6. □

Denote by τ°_n the sequence obtained from τ_n by removing all its elements τ^i_n such that X^i_n is ignored or squashed.

Lemma 25. *Let* $\texttt{prepared} \leq X^i_m, X^i_n \leq \texttt{dequeued}$. *If* $m < n$ *and* T^j_n *is the* r^{th} *register provider of* T^i_n *in* τ°_n, *then* T^j_m *is the* r^{th} *provider of* T^i_m *in* τ°_m. *Also, if* T^i_n *does not have the* r^{th} *register provider in* τ°_n, *then* T^i_m *does not have the* r^{th} *provider of in* τ°_m.

Proof. We prove only the first assertion of the lemma. The proof of the second is analogous.

Since $X_m^i \leq X_n^i$ and $X_m^j \leq X_n^j$, neither of X_m^i, X_m^j is `ignored` or `squashed`, so T_m^i, T_m^j are in τ_m°. By Lemma 18, $\mathsf{rSource}_r(T_m^i) = \mathsf{rSource}_r(T_n^i)$ and $\mathsf{rDest}(T_m^j) = \mathsf{rDest}(T_n^j)$. Thus, $\mathsf{rSource}_r(T_m^i) = \mathsf{rDest}(T_m^j)$.

Suppose now k is such that $j < k < i$, T_m^k is in τ_m°, and $\mathsf{rSource}_r(T_m^i) = \mathsf{rDest}(T_m^k)$. Again, by Lemma 18, we have $\mathsf{rSource}_r(T_n^i) = \mathsf{rDest}(T_n^k)$, so T_n^k does not belong to τ_n°. Thus, X_n^k is `ignored` or `squashed`. The only possibility for $X_n^k =$ `ignored` would be that $m = n-1$ and $X_m^k =$ `fetched`, but that contradicts Lemma 17. If $X_n^k =$ `squashed`, then it would follow that there exists p such that T_p^k belongs to $\mathtt{squashed}^p$. This would imply that T_p^i belongs to $\mathtt{squashed}^p$, which is not true. □

Proof of Proposition 3. Suppose T and U are transactions in τ_∞^{D} such that U is the r^{th} register provider of T. Let i and j be the ordinals of T and U respectively. Denote $H_k^i = (T_k, X_k)$ and $H_k^j = (U_k, Y_k)$. Let $n = \mathtt{sr}(i)$ and let m be the unique integer such that $X_m =$ `prepared`. From Lemma 11 we have that every k such that $m \leq k < n$ is regular.

By Lemma 16, U_n is the r^{th} provider of T_n in τ_n^{D}. Then, by Lemma 25, U_m is the r^{th} provider of T_m in τ_m°. By Lemma 17, $Y_m \geq$ `prepared`. Note also that, being an element of $\mathtt{prepared}^m$, T_m belongs to $\mathtt{queue}^m$.

Case 1: $Y_m =$ `dequeued`. By Lemma 17, T_m does not have an r^{th} register provider in $\mathtt{active}^m$. It follows, using Lemma 10, that T_m does not have an r^{th} provider in $\mathtt{queue}^m$. Let $\mathtt{reg} = \mathsf{rSource}_r(T_m)$. By Ord-Inv 4, $\mathtt{reg} \neq \bot$ and $\mathsf{rOp}_r(T_m) = \mathtt{rf}^m(\mathtt{reg})$. Since U_m is the r^{th} provider of T_m in τ_m°, it follows that U_m is the last transaction in τ_m^{D} whith rDest field equal to `reg`. It follows from Lemma 24 that $\mathsf{rOp}_r(T_m) = \mathsf{rRes}(U_m)$ and so $\mathsf{rSources}(T) = \mathsf{rRes}(U)$, as required.

Case 2: $Y_m \neq$ `dequeued`. Now Y_m belongs to $\mathtt{active}^m$. Since U_m is the r^{th} provider of T_m in τ_m°, it follows that U_m is the r^{th} provider of T_m in $\mathtt{active}^m$ as well. From Lemma 10 we deduce that U'_m is the r^{th} provider of T_m in $\mathtt{queue}^m$, where $U'_m \preceq U_m$. It follows from Ord-Inv 4 that $\mathsf{rOp}_r(T_m) = \bot$ and $\mathsf{rProv}_r(T_m) = \mathsf{name}(U'_m)$, which immediately implies $\mathsf{rProv}_r(T_m) = \mathsf{name}(U_m)$.

Since $T_m \preceq \cdots \preceq T_n$, $\mathsf{rOp}(T_m) = \bot$ and $\mathsf{rOp}(T_n) \neq \bot$, there exists a unique number p such that $m \leq p < n$, $\mathsf{rOp}(T_p) = \bot$, and $\mathsf{rOp}(T_{p+1}) \neq \bot$. Since T_p is incomplete, it belongs to $\mathtt{executing}^p$ or $\mathtt{prepared}^p$. From Lemma 9 we conclude that T_{p+1} is the descendant of T_p. Furthermore, Exec-Inv 5 implies that there exists a transaction V in $\mathtt{executing}^p \cup \mathtt{lingering}^p$ such that $\mathsf{rProv}_r(T_p) = \mathsf{name}(U)$ and $\mathsf{rOp}_r(T_{p+1}) = \mathsf{rRes}(V) \neq \bot$. It follows that $\mathsf{name}(V) = \mathsf{name}(U_m)$. We claim that $V = U_p$, which then implies $\mathsf{rOp}_r(T) = \mathsf{rOp}_r(T_{p+1}) = \mathsf{rRes}(U_p) = \mathsf{rRes}(U)$, finishing the proof.

Suppose the claim is not true. Then U_p cannot belong to $\mathtt{active}^p$ because this sequence contains V and cannot contain two transactions with the same name. It follows that $X_p^j =$ `dequeued`, so there exists q such that $p < q \leq m$ and U_q is in $\mathtt{computed}^q$ and so in $\mathtt{lingering}^q$. Since T_q belongs to $\mathtt{executing}^q$

and $\mathsf{rProv}_r(T_q) = \mathsf{name}(U_q)$, it follows that T_q depends on U_q. This contradicts Exec-Inv 3, finishing the proof of the claim.

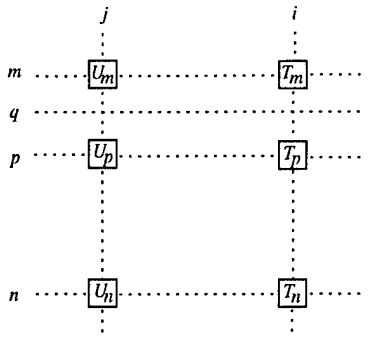


Fig. 8. Transactions involved in the resolution of a register dependency (Case 2 of the proof of Proposition 3).

We also need to prove $\mathsf{rOp}_r(T) = \mathtt{rf}_{init}(\mathsf{rSource}_r(T))$ in the case when $T = T^i_\infty$ does not have the r^{th} register provider in τ^{D}_∞. Let again m be the integer such that $X_m = \mathtt{prepared}$. By Lemma 25, T^i_m does not have the r^{th} provider in τ°_m. As in Case 1 above, we obtain $\mathsf{rOp}_r(T^i_m) = \mathtt{rf}^m(\mathtt{reg})$, where $\mathtt{reg} = \mathsf{rSource}_r(T^i_m)$. Using Lemma 24, we deduce $\mathtt{rf}^m(\mathtt{reg}) = \mathtt{rf}_{init}(\mathtt{reg})$, finishing the proof.

A.8 Proof of Proposition 4

Lemma 26. *Every load in* $\mathtt{prepared}^n + \mathtt{executing}^n$ *satisfies the condition* (LC).

Proof. By Ord-Inv 4, the mOp field of every load in $\mathtt{prepared}^n$ is $\perp$, so (LC) is true for such loads. Furthermore, every load in $\mathtt{executing}^n$ is a descendant of a load in $\mathtt{prepared}^{n-1} \cup \mathtt{executing}^{n-1}$, so by induction on n and using Exec-Inv 4, it follows that these loads also satisfy (LC). □

Lemma 27. *If* $\mathtt{computed}^n$ *contains a store, then this store is the first transaction in* $\mathtt{active}^n$.

Proof. Suppose the lemma is not true and pick the minimal n that provides a counter-example. Suppose T^i_n is a store in $\mathtt{computed}^n$ and T^j_n is the first transaction in $\mathtt{active}^n$, and $j < i$. Pick i so that $i - j$ is smallest.

Let U be the first transaction of $\mathtt{queue}^n$. By Lemma 10 $U \preceq T^j_n$. By By Exec-Inv 6, $\mathtt{writemem}^n = \mathrm{TRUE}$ and by Ord-Inv 8, U is an incomplete store. Using Lemma 17, we conclude that T^j_n belongs to $\mathtt{executing}^n$.

Let m be such that T^i_m belongs to $\mathtt{prepared}^m$. By Lemma 19, T^j_m belongs to $\mathtt{active}^m$ and so, by Ord-Inv 4, $\mathsf{mrSt}(T^i_m) = \mathsf{name}(T^k_m)$ for some k such that

$j \leq k < i$. Since $\mathsf{mrSt}(T_n^i) \neq \mathsf{mrSt}(T_m^i)$, it follows from Exec-Inv 7 that for some p such that $m < p \leq n$ one has T_p^k in $\mathtt{computed}^p$.

Since T_n^j is in $\mathtt{executing}^n$, T_p^j is not in $\mathtt{computed}^p$, so $k \neq j$. Thus, T_p^k is a store in $\mathtt{computed}^p$ and is not a first transaction in $\mathtt{active}^p$. By minimality of n, we have $p = n$ and then a contradiction with the minimality assumption on i. □

Corollary 28. *If* $\mathtt{active}^n$ *contains a complete store, then it is its first transaction. If* $\mathtt{dequeued}^n$ *contains a store, then it contains only one and it is its first transaction.* □

Proof. The first statement is an immediate consequence of Lemma 28. For the second, use also Lemma 13 □

Corollary 29. *If* $\mathtt{mem}^{n+1} \neq \mathtt{mem}^n$ *then the first transaction* S *in* $\mathtt{active}^n$ *is a complete store in* $\mathtt{executing}^n$ *and* $\mathtt{mem}^{n+1} = \mathtt{mem}^n \cdot S$.

Proof. The proof follows directly from Exec-Inv 6 and Corollary 28. □

Lemma 30. *If* $\mathtt{computed}^n + \mathtt{ripe}^n$ *contains a store* S, *then* $\mathtt{mem}^n = \mathtt{mem}^n \cdot S$.

Proof. Suppose $S = T_n^i$. Then, for some $m \leq n$, $T_m^i = S$ is in $\mathtt{computed}^m$, and so, by Exec-Inv 6, $\mathtt{mem}^m = \mathtt{mem}^{m-1} \cdot S$. Therefore, $\mathtt{mem}^m \cdot S = \mathtt{mem}^m$. For every p such that $m < p \leq n$, T_p^i is in $\mathtt{ripe}^p$ and is the first transaction in $\mathtt{active}^p$. It follows from Corollary 29, that $\mathtt{mem}^p = \mathtt{mem}^m$ for all such p. In particular, $\mathtt{mem}^n = \mathtt{mem}^m$ and the lemma follows. □

Lemma 31. *For every* n, $\mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau_n^{\mathsf{D}}$ *or* $\mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau_n^{\mathsf{D}} \cdot S$, *where* S *is a store and is a first transaction of* $\mathtt{active}^n$.

Proof. We argue by induction. The initial case is clearly true. For the induction step, suppose first that $\mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau_n^{\mathsf{D}}$. If $\mathtt{dequeued}^{n+1}$ is non-empty and contains no store, then $\mathtt{mem}^{n+1} = \mathtt{mem}^n$ by Lemma 29, and $\mathtt{mem}_{init} \cdot \tau_{n+1}^{\mathsf{D}} = \mathtt{mem}_{init} \cdot \tau_n^{\mathsf{D}} \cdot \mathtt{dequeued}^{n+1} = \mathtt{mem}_{init} \cdot \tau_{n+1}^{\mathsf{D}}$ is clear. If $\mathtt{dequeued}^{n+1}$ contains a store S, then by Lemma 28, $\mathtt{dequeued}^{n+1}$ begins with S and contains no other stores. Being an element of $\mathtt{dequeued}^{n+1}$, S belongs to $\mathtt{computed}^n$ or $\mathtt{ripe}^n$ (Eq. 12), so by Lemma 29, $\mathtt{mem}^{n+1} = \mathtt{mem}^n$. On the other hand, Lemma 30 implies $\mathtt{mem}^n = \mathtt{mem}^n \cdot S$ and so $\mathtt{mem}^n = \mathtt{mem}^n \cdot \mathtt{dequeued}^{n+1} = \mathtt{mem}_{init} \cdot \tau_{n+1}^{\mathsf{D}}$. Finally, if $\mathtt{dequeued}^{n+1}$ is empty, then $\tau_{n+1}^{\mathsf{D}} = \tau_n^{\mathsf{D}}$ and both $\mathtt{mem}^{n+1} = \mathtt{mem}^n$ and $\mathtt{mem}^{n+1} \neq \mathtt{mem}^n$ are possible. The desired result in the first case follows immediately, and in the second case it follows from Lemma 29.

Assume now the second possibility for the inductive hypothesis: $\mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau_n^{\mathsf{D}} \cdot S$, where S is a store and is a first transaction of $\mathtt{active}^n$. Lemma 29 implies $\mathtt{mem}^{n+1} = \mathtt{mem}^n$. If $\mathtt{dequeued}^{n+1}$ is empty, the result immediately follows. If $\mathtt{dequeued}^{n+1}$ is non-empty, then it begins with S and contains no other stores, so $\mathtt{mem}^{n+1} = \mathtt{mem}^n = \mathtt{mem}_{init} \cdot \tau_n^{\mathsf{D}} \cdot S = \mathtt{mem}_{init} \cdot \tau_n^{\mathsf{D}} \cdot \mathtt{dequeued}^{n+1} = \mathtt{mem}_{init} \cdot \tau_{n+1}^{\mathsf{D}}$. □

Lemma 32. *If T is a load or store in* $\texttt{prepared}^n + \texttt{executing}^n$ *and U is the most recent store of T in* $\texttt{active}^n$*, then either (1) U is in* $\texttt{prepared}^n + \texttt{executing}^n$ *and* $\mathsf{mrSt}(T) = \mathsf{name}(U)$*, or (2) U is in* $\texttt{computed}^n + \texttt{ripe}^n$ *(and therefore is the first transaction in* $\texttt{active}^n$*) and* $\mathsf{mrSt}(T) = \text{NONE}$.

Proof. Let $T_n^i = T$ and $T_n^j = U$. Let m be such that T_m^i is in $\texttt{prepared}^m$. Then T_m^j is in $\texttt{active}^m$ and so $\mathsf{mrSt}(T_m^i) = \mathsf{name}(T_m^k)$, where T_m^k is the most recent store for T_m^i in $\texttt{active}^m$. We claim that $k = j$. Otherwise, using Exec-Inv 7 we would obtain T_p^k in $\texttt{computed}^p$ for some $p \leq n$, contradicting Corollary 28. The lemma now follows from Exec-Inv 7 and Corollary 28. □

Corollary 33. *Let T be a load or store in* $\texttt{prepared}^n + \texttt{executing}^n$ *and let ϕ be the store chain of T in this set. Then ϕ is equal to the sequence of stores in* $\texttt{prepared}^n + \texttt{executing}^n$ *that precede T in* $\texttt{active}^n$. □

Lemma 34. *Let L be a load in* $\texttt{prepared}^n + \texttt{executing}^n$ *and let ψ be the prefix of* $\texttt{active}^n$ *consisting of transactions preceding L. Then* $\mathsf{mOp}(L) \preceq (\texttt{mem}^n \cdot \psi)(\mathsf{mSource}(L))$.

Proof. By Lemma 26, $\mathsf{mOp}(L) \preceq (\texttt{mem}^n \cdot \phi)(\mathsf{mSource}(L))$, where ϕ is the store chain of L in $\texttt{prepared}^n + \texttt{executing}^n$. Let ψ_0 be the sequence obtained by deleting from ψ all transactions which are not stores. Clearly, $\texttt{mem}^n \cdot \psi = \texttt{mem}^n \cdot \psi_0$. By Corollary 28, all transactions in ψ_0 are in $\texttt{prepared}^n + \texttt{computed}^n$, except possibly the first store (say, S), which may belong to $\texttt{computed}^n + \texttt{ripe}^n$. By Lemma 33, we have $\psi_0 = \phi$ in the first case, and $\psi_0 = \langle S \rangle \,\#\, \phi$ in the second. By Lemma 30, $\texttt{mem}^n \cdot \phi = \texttt{mem}^n \cdot \psi_0$, finishing the proof. □

Lemma 35. *Let α and β be transaction sequences such that $\alpha \preceq \beta$. Let* mem *be an element of type* **IAddr** $\rightarrow$ **Value** *and* addr *an element of type* **IAddr**. *Then* $(\texttt{mem} \cdot \alpha)(\texttt{addr}) \preceq (\texttt{mem} \cdot \beta)(\texttt{addr})$.

Proof. By direct examination. □

Proof of Proposition 4. Suppose L is a load in τ_∞^{D}. Let θ be the prefix of τ_∞^{D} consisting of transactions that precede L. We will prove that $\mathsf{mOp}(L) = (\texttt{mem}_{init} \cdot \theta)(\mathsf{mSource}(L))$. It is easy to see that this would imply Proposition 4.

Let i and be such that $L = T_\infty^i$ and and let n be the largest number such that T_n^i is in $\texttt{prepared}^n + \texttt{executing}^n$. Thus, T_{n+1}^i is in $\texttt{computed}^{n+1}$ and it follows from Exec-Inv 7 that $\mathsf{mOp}(T_n^i) \neq \bot$. Thus, $\mathsf{mOp}(T_n^i) = \mathsf{mOp}(L)$ and $\mathsf{mSource}(T_n^i) = \mathsf{mSource}(L)$.

From Lemma 34 we then obtain $\mathsf{mOp}(L) \preceq (\texttt{mem}^n \cdot \psi)(\mathsf{mSource}(L))$, where ψ is the prefix of $\texttt{active}^n$ consisting of transactions preceding L. By Lemma 31, $\texttt{mem}^n$ is equal to either $\texttt{mem}_{init} \cdot \tau_n^{\mathsf{D}}$ or $\texttt{mem}_{init} \cdot \tau_n^{\mathsf{D}} \cdot S$, where the store S is the first transaction of $\texttt{active}^n$. Since ψ is a prefix of $\texttt{active}^n$ (and L is not a store), it follows that $\texttt{mem}^n \cdot \psi = \texttt{mem}_{init} \cdot \tau_n^{\mathsf{D}} \cdot \psi$.

By Lemma 20, all transactions of ψ are eventually dequeued. Thus, $\tau_n^{\mathsf{D}} \,\#\, \psi \preceq \theta$. Using Lemma 35, we finally obtain $\mathsf{mOp}(L) \preceq (\texttt{mem}_{init} \cdot \theta)(\mathsf{mSource}(L))$, which must be equality because $\mathsf{mOp}(L) \neq \bot$.

On embedding a microarchitectural design language within Haskell

John Launchbury, Jeff Lewis and Byron Cook
Oregon Graduate Institute

Abstract

Based on our experience with modelling and verifying microarchitectural designs within Haskell, this paper examines our use of Haskell as host for an embedded language. In particular, we highlight our use of Haskell's lazy lists, type classes, lazy state monad, and `unsafePerformIO`. We also point to several areas where Haskell could be improved.

1 Introduction

There are many ways to design and implement a language — not all of them imply building from the ground up. Landin's vision of the next 700 programming languages [18], for example, was to build domain-specific vocabularies on top of a generic language substrate. In the verification community, this is known as a *shallow embedding* of one language or logic into another. From our programming language perspective we believe that, in effect, every abstract type defines a language. Admittedly, most abstract types by themselves make poor languages, but when interesting combinators are provided the language suddenly becomes rich and vibrant in its own right. This explains the continuing popularity of combinator libraries, from the time of Landin until now.

The animation language/library Fran is a beautiful example [10, 9]. Fran provides two families of abstract types in Haskell: behaviors and events. To construct a term of type `Behavior Int`, for example, is to write a sentence in the Fran language, using Fran primitives and Fran combinators. To build complex Fran entities, however, the full power of Haskell can be brought to bear. Fran objects are just another abstract data type.

How good is Haskell at hosting other languages? This is one of those questions that can only be answered through experience—and is precisely where we can contribute. In this paper we describe our use of Haskell as a host to a microarchitectural modelling language, calling attention to the aspects of Haskell that helped us, those that hindered us, and the features we wish we had. In particular, we highlight our use of Haskell's lazy lists, type classes [15], the lazy state monad [19], and `unsafePerformIO` [17]. This paper contains no deep theory, but rather a dose of measured introspection.

The remainder of this paper is organized as follows: In Section 2 we provide the motivation to our work in microarchitectural modelling. In Section 3 we introduce Hawk and show how we use lazy lists to model wires. In Sections 4, 5, and 6, we show how type classes, the lazy state monad, and `unsafePerformIO`, respectivly, are put to use in Hawk. In Section 7 we describe an application that makes use of all four features. In the final sections we outline where Haskell has constrained us, and discuss future work.

2 Building a microarchitectural description language

Contemporary superscalar microarchitectures employ tremendously aggressive strategies to mitigate dependencies and memory latency. Their complexity taxes current design techniques to the limit. The trend continues, as the size of design teams grows exponentially with each new generation of chip.

To gain an appreciation for the complexity of modern microarchitectures, take as an example the model of an instruction reorder buffer (ROB) which occurs frequently in out-of-order microprocessors like the Pentium III. The function of the ROB is to maintain a pool of instructions, and to determine dynamically which of them are eligible for delivery to an execution unit once their operands have been computed. This way, instructions are executed at the earliest possible moment. Furthermore, instructions are introduced speculatively, based upon numerous successive branch predictions. Consequently, instructions that have previously been scheduled and executed must sometimes be rescinded when a branch is discovered to have been mispredicted. Thus the ROB must keep track of instructions up to the point that they can either be retired (committed) or flushed.

Since some instructions following a branch may already have been executed when a branch misprediction is discovered, register contents are also affected. At a branch misprediction, register mapping tables must be modified to invalidate the contents of registers that contain results of rescinded instructions. The contents of registers that are possibly live must be preserved until after the branch has been resolved, thus increasing the complexity of the interaction between a ROB and the registers.

In addition, there are all the issues of managing on-chip resources, of ensuring rapid and correct communication of results, of cache coherence and so on. It will get worse. The next generation of microarchitectures will address many more issues such as explicit instruction parallelism [13] and multiple instruction threads [29].

As if all these algorithms did not provide enough design complexity, commercially viable microarchitectures are also subject to legacy requirements. For example Intel's Pentium III must deal with dozens of exception types to remain compatible with earlier versions of the X86 archi-

tecture. Pentium III also struggles with the variable length of X86 instructions. It tries to fetch three each cycle, and it turns out that dynamically determining the length of instructions before decoding is one of Pentium III's primary performance bottlenecks. Again, this type of problem is not going to go away. Intel's upcoming Merced processor will execute not only its new instruction set [8], but X86 as well [12].

With designs of this complexity, it is hard to imagine that designers will not stumble upon subtle concurrency bugs. The need for powerful and effective modelling and verification has never been greater. By couching microarchitecture modelling in terms of higher-level abstractions and emphasizing the modularity of a design it is possible to regain control of the design space. This is what we have done. In conjunction with Intel's Strategic CAD Laboratory, we have developed *Hawk* as an executable modelling language embedded in Haskell. Hawk is very high level compared with other hardware description languages. Consequently, even complex microarchitecture models remain remarkably brief, allowing designers to retain a high level of intellectual control over the model. For example, the complete formal model of a speculative, superscalar, out-of-order microarchitecture based on the Pentium III required less than 1000 lines of code [5].

3 Lazy lists: adding signals to Haskell

Effectively, Hawk is an embedding of Lustre-style signals [4] into Haskell. Signals model values that change over time, like wires in a microprocessor. Following O'Donnell [24], Srivas & Bickford [28], and many others, we implement signals as lazy lists. The idea is very simple: the n^{th} element of the list represents the value of the wire at clock tick n. Thus the value of each wire is a complete description of its behavior over time. This approach leads to circuit semantics with a definite denotational flavor. In contrast, state transition systems (another popular style) are much more operational in their nature. There are naturally advantages and disadvantages to each.

To represent units with clocked inputs and clocked outputs we use functions from signals to signals, known as list transformers (or stream transformers). Combinational circuits can be turned into clocked circuits simply by mapping them down their input lists. So if `add::(Int,Int)->Int` acts like a simple addition circuit, then `map add :: [(Int,Int)] -> [Int]` is its clocked equivalent.

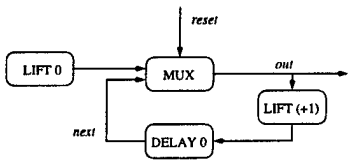
The fundamental non-combinational circuit is the *delay*. The delay is what makes feedback loops in clocked circuits possible—without any delays, a feedback loop would just generate smoke! A delay is defined so that the $(n+1)^{st}$ element of the output is equal to the n^{th} element of its input, with an initial value output for the very first clock tick. The implementation of `delay :: a -> [a] -> [a]` is simply "cons".

Some care is needed within this paradigm, however. Arbitrary use of list processing functions, especially those which discard elements, such as filter, can cause problems in that they may require infinite buffers to implement. To restrict the way in which a signal can be constructed or altered, we make the signal type abstract in Hawk and provide a basic set of manipulation functions that are known to be safe.

```
newtype Signal a

delay :: a -> Signal a -> Signal a
lift0 :: a -> Signal a
lift1 :: (a -> b) -> Signal a -> Signal b
 .    ::    .
 .    ::    .
 .    ::    .
```

`lift0` returns a constant signal; and `lift1` is just map. Later we will use the derived operator `bundle`, which takes a pair of signals, and produces a signal of pairs. Restricting access to the implementation in this way gives the usual freedoms to provide alternative implementations, or even to refine the semantics somewhat. For example, we could implement signals as functions from the natural numbers to values.

If the above signature seems to be missing something — it is. The rest comes from Haskell itself, in particular, lazy recursive definitions. You could say that the missing operator of the abstract type is a (lazy) fixpoint operator. Consider a resettable counter circuit like:



which, in Hawk, we might model as:

```
counter reset = out
  where
  next = delay 0 (lift1 (+1) out)
  out  = mux reset (lift0 0) next
```

Note the mutual recursion between signals. The laziness of Haskell is vital for this definition to have the intended meaning.

One thing that is not missing is a way to observe a list by taking its head or tail. This is intentional. A circuit that was specified to take the tail of a list would be asking for an infinite buffer. We *do* allow signals to be viewed as lists for the purpose of viewing simulation results, but this operation is only provided for use at the top-level.

4 Organizing microarchitectural abstractions with type classes

The point of Hawk has been to build abstractions that increase the concision of microarchitectural models [5], and facilitate the verification process [22].

In order for microarchitectural abstractions to be relevant, they must be extraordinarily flexible in the types that they operate over. Instruction sets differ in variety of details: size and type of data, number and types of registers, and the instructions themselves. Internally, machines may use other instruction sets. For example, the AMD K6[27] implements the X86 instruction set, but uses a RISC instruction set within its execution core.

We use type classes to facilitate the description of circuits that operate over all instruction sets. For example, the type of an ALU might be:

```
alu :: (Instruction i, Bits w) => (i,w,w) -> w
```

This way `alu` can be used in a X86 model (where `w` is set to 32-bit words and `i` to X86 instructions) or a 64-bit RISC instruction set, like that of the Alpha. The `Bits` class is an extension of Haskell's `Num` class that adds operators related to word size, signedness, etc. The `Instruction` class captures the common elements between different instructions sets.

With common architectural characteristics captured with type classes, we are then able to build abstractions that help organize microarchitectural models. For example, *transactions* [1, 23] are a simple yet powerful grouping of control and data. A transaction is a machine instruction grouped together with its state. This state might include:

- Operand values.
- A flag indicating that the instruction has caused an exception.
- A predicted jump target, if the instruction is a branch.

Microarchitectures models that utilize transactions can then make decisions locally rather than with a seperate control unit.

Hawk provides a library of functions for creating and modifying transactions. For example, `bypass` takes two transactions and builds a new transaction where the values from the destination operands of the first transaction are forwarded to the source operands of the second. If `i` is the transaction:

```
(r4,8) <- (r2,4) + (r1,4)
```

and `j` is the transaction:

```
r10 <- (r4,6) + (r1,4)
```

then `bypass i j` produces the transaction:

```
r10 <- (r4,8) + (r1,4)
```

That is, `bypass` inserted `i`'s more recent valuation of `r4` into the destination operand of `j`.

By parameterizing over the instances of finite words and registers:

```
bypass :: (Bits w, Register r) =>
          Trans i r w -> Trans i r w -> Trans i r w
```

`bypass` can be used in many contexts. Within our Pentium III-like microarchitectural model we use `bypass` on both instructions with real register references and virtual register references (both are instances of the type class `Register`). In our Merced-like model [6], we use the same `bypass` with IA-64 instructions.

5 Lazy state: using state-based components

There has been debate in the Haskell community about the merits of strictness within the state monad. In this section we describe an application where a lazy state monad is the right thing.

Some microarchitectural components, such as register files, are more naturally (and effeciently) presented as state transition systems than list transformers. Fortunately, we can easily embed state-based models into the list transformer idiom using the *lazy* state monad and `runST` [19].

Imagine modelling a register file as an array which, on each clock tick, is both written to and read from.

```
reg :: Register r => Signal (r,w) -> Signal r ->
                     Signal w
reg writes reads
  = runST (
    do { reg <- newArray (minAddr, maxAddr) init
       ; loopST (regFile reg) (bundle writes reads)
       }
    )

regFile :: STArray s Addr Val -> ((Addr,Val), Addr)
           -> ST s Val
regFile reg ((a,w),r)
   = do { writeArray reg a w
        ; readArray reg r
        }
```

where `loopST` is a monadic map on signals:

```
loopST :: (a -> ST s b) -> Signal a
          -> ST s (Signal b)
```

The semantics of lazy state is as follows. The monadic structure sequentializes the operations of the monad but *forces nothing*. As the result of the state thread is demanded, so execution proceeds, but in the order determined by the monadic sequentialization. Thus execution proceeds on demand, but some of that demand is transmitted through the state sequencer.

The state within the scope of `runST` is completely hidden from the outside world. Thus as far as the rest of the program is concerned, `reg` is completely pure, as indicated by its type. The encapsulation of the state occurs because of the type of `runST`. Inside the implementation of `regFile`, however, the situation is quite different. The array writes are "imperative", having effects immediately visible to subsequent reads.

In the use of `loopST` above, the state machine is executed step by step, consuming its list input and generating its list output on the way. In particular, the `loop` construct did not attempt to execute the state machine completely before releasing the output list. It is this behavior we *require* of the state monad and, fortunately, though not officially a part of Haskell, most implementations provide it.

6 Monitoring circuits with `unsafePerformIO`

When embedding a language, one often needs "language primitives" that provide good things in bad ways. Fran for example, has a function :

```
importBitmap :: Filename -> Bitmap
```

which imports a bitmap file in the `IO` monad but uses `unsafePerformIO` to treat the bitmap as a pure value.

When using Hawk we find that one often wants to observe the values flowing across a signal. Unfortunately, Haskell's semantic purity makes this viewing rather difficult. Often, without re-coding a model, it is not possible to observe the signal. Therefore we provide the function:

```
probe :: Filename -> Signal a -> Signal a
```

As far as Hawk-level models are concerned, a probe is simply an identity. However, the external world receives a different view. Probes are fundamentally side-effecting, writing values to a file, even though they apparently have a pure type. Thus probes cannot be defined within Haskell-proper. Instead, they required some Haskell system hacking through the use of `unsafePerformIO`.

```
probe name vals = zipWith (write name) [1..] vals

write name clock val = unsafePerformIO
   do { h <- openFile name AppendMode
      ; hPutStrLn h (show clock ++ ":" ++ outp val)
      ; hClose h
      ; return val
      }
```

Notice that we are careful not to change the strictness of lazy lists.

We have found that `unsafePerformIO` is a powerful facility for building of domain-specific tools that observe, but do not affect the microarchitectural models.

7 Verification in Hawk

The past several sections have, one-by-one, demonstrated the usefulness of lazy lists, type classes, the state monad, and `unsafePerformIO`. In this section we discuss a particularly exciting application that requires all four features.

Hawk provides tools that can be used to formally verify properties of models. Suppose that we want to prove the following properties about the resettable counter from Section 3:

1. when the reset line is low on the next clock cycle, the output is the value at the current cycle plus 1,
2. and when the reset line is high at the current clock cycle, the output is zero.

In Hawk, we might express these properties as follows. Assume that `r0` and `r1` are the values of the reset line at time t and $t+1$ respectively, and that n and m are the corresponding outputs.

```
prop_counter = prop_one && prop_two
  where
    prop_one = not r1 ==> (n + 1 === m)
    prop_two = r0 ==> (n === 0)
```

The trick is to show that these properties hold for arbitrary values of `r0` and `r1`. To do that, we will use symbolic values for `r0` and `r1`, and symbolically simulate the circuit.

The approach we take to symbolic simulation [7] is straightforward. Take a sufficiently polymorphic function, and instantiate it at a symbolic datatype. What we mean by a symbolic datatype is any datatype that is enriched with variables and additional term structure. For example, we have used the following datatype for symbolic simulation of simple arithmetic circuits.

```
data Symbo a =
       Const a
     | Var String
     | Plus (Symbo a) (Symbo a)
     | Times (Symbo a) (Symbo a)
```

The catch is that some care is required in making functions "sufficiently" polymorphic. This means that over the parts of the program that you wish to symbolically evaluate, you cannot use concrete types, because those types must be able to become symbolic.

7.1 Fitting symbolic simulation into Haskell

In places, such as with the `Num` class, Haskell's prelude is remarkably amenable to symbolic simulation. In others it is not. As an example, consider Booleans. To capture the operations of both concrete and symbolic Booleans we have defined a class `Boolean`, which makes all the boolean operators from the prelude abstract:

```
class Boolean b where
    true :: b
    false :: b
    (&&) :: b -> b -> b
    (||) :: b -> b -> b
    (==>) :: b -> b -> b
    not :: b -> b
```

We have also defined the class `Eql`, which is like the standard `Eq` class, except that it is also abstracted over the result type for equality, resulting in a multi-parameter type class:

```
class Eql a b where
    (===) :: a -> a -> b
```

Conditional expressions, too, must be abstract:

```
class Mux c a where
    mux :: c -> a -> a -> a
```

If the condition on which we branch is symbolic, then it is clear that the result must be symbolic as well. Hence there is a relationship between the type of the conditional, and the type of the result—just the sort of thing that multi-parameter type classes express well.

To capture the common usage of conditional expressions, we make `Bool` an instance of `Mux`

```
instance Mux Bool a where
    mux x y z = if x then y else z
```

We can now employ many implementations of Booleans. In particular we can use binary decision diagrams (BDDs) [3], which implement semantic equality between symbolic boolean expressions in constant time. Using H/Direct [11], the state monad and `unsafePerformIO`, we have imported the CMU BDD package into Haskell. In the style of the modelling language of Voss [26], Hawk treats BDDs just like Booleans. But, thanks to type classes, a user can also choose *not* to use BDDs — so long as their choice is an instance of `Boolean`.

7.2 Proving a property

We now have the infrastructure to verify our properties. Our strategy is to simulate the counter with symbolic values on the reset line for the first two ticks, and then test the desired property on the first two outputs. We have made the initial value of the delay in the counter an additional parameter so that we can place a symbolic value there as well. This makes our test independent of the internal state of the counter, and thus makes it valid to test the properties only at the first two clock ticks.

```
test :: BDD
test = prop_one && prop_two
    where
    a = var "a" :: BDD_Vector8
```

```
r0 = var "r0" :: BDD
r1 = var "r1" :: BDD
reset :: Signal BDD
reset = r1 'delay' r1 'delay' false
[n, m] = counter a reset @@@ [0, 1]
prop_one = not r1 ==> (n + 1 === m)
prop_two = r0 ==> (n === 0)
```

(`@@@` is an operator for sampling a signal at the specified times.) By evaluating `test` we are proving that, for Boolean vectors of length 8, the `counter` circuit meets our specification. Using types more general that `BDD_Vector8`, we can prove the properties for counters of arbitrary size.

8 Where Haskell and Hawk tangle

For our domain, Haskell has turned out to be an excellent tool for experimenting with language design. However, in a few places, Haskell is not a perfect match. In this section we review our use of lazy lists, type classes, the lazy state monad, and `unsafePerformIO` and point to the hinderences that we have encountered.

8.1 Lazy Lists

In some cases Haskell is a little too generous. Our preferred semantics for signals is that of truly infinite, or coinductive, lists—i.e., not that of finite, infinite, and partially defined lists, as in Haskell. Any feedback loop that did not include at least one delay should be rejected as being ill-defined. Haskell, however, will stubbornly do its best to make sense of even such ill-defined definitions. Could Haskell do better? We have constructed a shallow embedding of Hawk in Isabelle [25], which is much less forgiving. In order to have Isabelle accept our recursive definitions we have had to develop a richer theory of induction over coinductive datatypes than previously available [21]. Using this theory, Isabelle is able to accept all the valid Hawk definitions that we have thrown at it, while rejecting the invalid ones. It would be useful if Haskell's type system could be extended to handle this—perhaps using unpointed types [20] to express valid coinductive definitions.

8.2 Type Classes

Because the type representing an instruction set must remain abstract, we cannot directly pattern match on it. Instead, the operations of the `Instruction` class provide predicates to identify common instructions such as nops, arithmetic ops, loads and stores and jumps.

```
class (Show i, Eq i) => Instruction i where
    isNoOp :: i -> Bool
    isAddOp :: i -> Bool
    isSubOp :: i -> Bool
    ...
```

If Haskell allowed arbitrary views of datatypes [30], then this could be handled much more nicely.

8.3 The State Monad

Haskell's syntactic support for state is not a perfect fit. First, Haskell has no way to declare storage statically, but this is exactly what is required. In the register example, the array is allocated at the beginning, and nothing else is allocated afterwards. Since silicon cannot be allocated on the fly, when we come to consider other interpretations of Hawk models, it would be useful to guarantee that the body of the state code did not affect the shape of the store, merely its contents.

Secondly, in our microarchitectural models, the pattern `loopST f (bundle xs ys)` occurs often enough to want a language construct to describe it. Putting these ideas together, we may ideally wish to write something like:

```
reg writes reads
  = runST (do {array reg (minAddr, maxAddr) = init
              ; loop (w<-writes, r<-reads)
                   {  writeArray reg a w
                   ;  readArray reg r
                   }
               }
           )
```

8.4 Using `unsafePerformIO`

Probes often work quite well, but there are some glitches. While we have been careful to preserve the semantics of Haskell in introducing probes, the semantics of probes are not really preserved by Haskell. Due to lazy evaluation, there's nothing to assure that probe output will appear in the order expected. The output of a probe at clock tick 9 might be put in the file before the output of a probe at clock tick 7. Another, glitch is that, in a model, we are free to use a given unit more than once. But if that unit has an embedded probe, you will get the output of both probes in the file. This is not problematic, except that you have no way of identifying which output is from which probe.

But these problems have less to do with the perhaps unscrupulous nature of using `unsafePerformIO`, and more to do with a shortcoming in our overall design. In the section on future work, we will discuss an approach that will mitigate these problems.

8.5 Symbolic simulation

Our drive to make the entire Hawk library sufficiently polymorphic to perform symbolic evaluation has made us painfully aware of the shortcomings of Haskell's type class system in describing abstract data types. Haskell's module system can be used in a limited way to effect abstraction, as we have used for the signal type. But Haskell's module system is only intended as name space management, and is a poor match when you intend to use abstract types instantiated at many different types.

The type class system at times works brilliantly. And what is most impressive is how well it has worked for us, as we use it for tasks far beyond its original intended use (simply as a system of overloading). However, the fit is not always perfect. One place is the lack of explicit control over instancing. One of the neat aspects of symbolic evaluation is that it allows us to take an existing executable model and verify properties of it, without changing the model at all. However, this does not work quite as well as it could because of limitations in the class system. Ideally, we would like to instantiate `test` above at different symbolic types. However, there is no good way to parameterize `test` by the types in question, without resorting to unpleasantries like adding dummy arguments. The type of the counter data

is purely an intermediate value in the definition of `test`. If we were not specific about the type of `a`, Haskell would consider the declaration ambiguous. Here we are limited by the type class system's restriction to type inference—the programmer is given no tool to resolve the ambiguity. Just as type inference can be augmented by type annotations to help the type system where it can't help itself, as with polymorphic recursion, we should be able to provide some sort of annotation to help Haskell resolve ambiguous uses of type classes.

9 Future work

9.1 Verification

One of the unsatisfying aspects of the verification example is that it was necessary to make the internal state of the counter an explicit parameter. Doing this in a complex model would entail passing around a lot of extra parameters—just the sort of thing we'd like to avoid. Also, in forcing the model to be explicit about its internal state, it also undercuts one of the strengths of the signal transformer model that sets it apart from state transformer models, making it a sort of unwelcome hybrid.

However, using ideas from Symbolic Trajectory Evaluation [14], we are currently working with symbolic domains that have a partial order structure. Symbolic simulation proceeds by starting with initial states set to bottom, with iteration of the model gradually adding more information.

We are also currently applying symbolic simulation to simple pipelined microarchitectures to verify correctness of hazard avoidance, using a self-consistency checking approach [16]. The technique is to simulate a stream of symbolic instructions two times. Let us assume that the pipeline has two stages. In the first case, we feed two symbolic instructions followed by a no-op. In the second case, we feed the same two symbolic instructions *separated* by the no-op. The test is that the contents of the registers is the same after the third instruction, demonstrating that the hazard logic is working correctly.

9.2 Elaboration monads

One of the shortcomings of Hawk is that it has no explicit notion of elaboration separate from the semantics of the model. Elaboration is the process of translating a possibly higher-order Hawk circuit into a first-order description, such as the hardware languages VHDL or Verilog. This was not always the case. Initially, Hawk was similar to Lava [2], using a monad to capture circuit elaboration. The monad might be used to generate net-lists for the purposes of fabrication, or it might produce logical formulae for input to a theorem prover. For simulation, the monad is essentially the identity monad, since all we have to do is glue together functions. However, during simulation, the monad could also provide the service of, for example, splitting probes that get duplicated.

One reason that we departed from an explicit monadic style is that the mutually recursive streams idiom that works so well is not supported by the `do` notation. What we propose is to extend the `do` notation so that bindings are recursive.

10 Acknowledgements

For their contributions we would like to thank Mark Aagaard, Borislav Agapiev, Todd Austin, Robert Jones, John O'Leary, and Carl-Johan Seger of Intel Corporation; Tim Leonard and Abdelillah Mokkedem of Compaq/Digital Corporation; Simon Peyton Jones of Microsoft Corporation; Per Bjesse, Koen Claessen, and Mary Sheeran of Chalmers; Elias Sinderson of GlobalStar; and Dick Kieburtz, John Matthews, Nancy Day, Sava Krstić, Thomas Nordin, Tito Autrey, and Mark Shields of OGI.

This research is supported in part by Intel, the National Science Foundation, the Defense Advanced Research Projects Agency, and Air Force Material Command.

References

[1] AAGAARD, M., AND LEESER, M. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Sept. 1994).

[2] BJESSE, P., CLAESSEN, K., SHEERAN, M., AND SINGH, S. Lava: Hardware design in Haskell. In *International Conference on Functional Programming* (Baltimore, July 1998).

[3] BRYANT, R. E. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys 24*, 3 (1992).

[4] CASPI, P., PILAUD, D., HALBWACHS, N., AND PLAICE, J. Lustre: A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages* (Munich, Germany, Jan. 1987).

[5] COOK, B., LAUNCHBURY, J., AND MATTHEWS, J. Specifying superscalar microprocessors with Hawk. In *Workshop on Formal Techniques for Hardware* (Maarstrand, Sweden, June 1998).

[6] COOK, B., LAUNCHBURY, J., MATTHEWS, J., AND KIEBURTZ, D. Formal verification of explicitly parallel microarchitectures, 1999. Submitted for publication.

[7] DAY, N. A., LEWIS, J. R., AND COOK, B. Symbolic simulation of microprocessor models using type classes in Haskell. Submitted for publication.

[8] DULONG, C. The IA-64 architecture at work. *IEEE Computer 31*, 7 (1998).

[9] ELLIOTT, C. An embedded modeling language approach to interactive 3D and multimedia animation. *To appear in IEEE Transactions on Software Engineering* (1999).

[10] ELLIOTT, C., AND HUDAK, P. Functional reactive animation. In *The International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997).

[11] FINNE, S., LEIJEN, D., MEIJER, E., AND JONES, S. P. H/Direct: A binary foreign language interface for Haskell. In *International Conference on Functional Programming* (Baltimore, July 1998).

[12] GWENNAP, L. First Merced patent surfaces. *Microprocessor Report 11*, 3 (1997).

[13] GWENNAP, L. Intel, HP make EPIC disclosure. *Microprocessor Report 11*, 14 (1997).

[14] HAZELHURST, S., AND SEGER, C.-J. H. Symbolic trajectory evaluation. In *Formal Hardware Verification.* Springer-Verlog, 1997.

[15] JONES, M. P. *Qualified Types: Theory and Practice.* PhD thesis, Department of Computer Science, Oxford University, 1992.

[16] JONES, R. B., SEGER, C.-J. H., AND DILL, D. L. Self-consistency checking. In *Formal Methods in Computer-Aided Design* (Palo Alto, California, 1996).

[17] JONES, S. P., AND MARLOW, S. Stretching the storage manager: weak pointers and stable names in Haskell, 1999. Submitted for publication.

[18] LANDIN, P. J. The Next 700 Programming Languages. *Communications of the ACM 9*, 3 (March 1966), 157–164.

[19] LAUNCHBURY, J., AND JONES, S. P. Lazy functional state threads. In *Programming Languages Design and Implementation* (Orlando, Florida, 1994), ACM Press.

[20] LAUNCHBURY, J., AND PATTERSON, R. Parametricity and unboxing with unpointed types. In *The International Conference on Functional Programming* (1996).

[21] MATTHEWS, J. Recursive function definition over coinductive types. Submitted for publication.

[22] MATTHEWS, J., AND LAUNCHBURY, J. Elementary microarchitecture algebra. In *International Conference on Computer-Aided Verification* (Trento, Italy, July 1999).

[23] MATTHEWS, J., LAUNCHBURY, J., AND COOK, B. Specifying microprocessors in Hawk. In *IEEE International Conference on Computer Languages* (Aug. 1998).

[24] O'DONNELL, J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education* (July 1995).

[25] PAULSON, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.

[26] SEGER, C.-J. Voss – a formal hardware verification system. Tech. Rep. 93-45, University of British Columbia, 1993.

[27] SHRIVER, B., AND SMITH, B. *The Anatomy of a High-Performance Microprocessor: A Systems Perspective.* IEEE Computer Society Press, 1998.

[28] SRIVAS, M., AND BICKFORD, M. Formal verification of a pipelined microprocessor. *IEEE Software 7*, 5 (1990).

[29] TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture* (Philadelphia, PA, May 1996).

[30] WADLER, P. Views: a way for pattern matching to cohabit with data abstraction. In *14'th ACM Symposium on Principles of Programming Languages* (Munich, Germany, January 1987).

Elementary Microarchitecture Algebra

John Matthews and John Launchbury

Oregon Graduate Institute,
P.O. Box 91000, Portland OR 97291-1000, USA
{johnm,jl}@cse.ogi.edu
http://www.cse.ogi.edu/PacSoft/Hawk

Abstract. We describe a set of remarkably simple algebraic laws governing microarchitectural components. We apply these laws to incrementally transform a pipeline containing forwarding, branch speculation and hazard detection so that all pipeline stages and forwarding logic are removed. The resulting unpipelined machine is much closer to the reference architecture, and presumably easier to verify.

1 Introduction

Transformational laws are well known in digital hardware, and form the basis of logic simplification and minimization, and of many retiming algorithms. Traditionally, these laws occur the gate level: de Morgan's law being a classic example. In this paper, we examine whether corresponding transformational laws hold at the microarchitectural level.

A priori, there is no reason to think that large microarchitectural components should satisfy any interesting algebraic laws, as they are constructed from thousands of individual gates. Boundary cases could easily remove any uniformity that has to exist for simple laws to be present. Yet we have found that when microarchitectural units are presented in a particular way, many powerful laws appear. Moreover, as we demonstrate in this paper, these laws *by themselves* are powerful enough to allow us to show equivalence of pipelined and non-pipelined microarchitectures.

We have used this algebraic approach to simplify a pipelined microarchitecture that uses forwarding, branch speculation and pipeline stalling for hazards. The resulting pipeline is very similar to the reference machine specification (i.e. no forwarding logic), while still retaining cycle-accurate behavior with the original implementation pipeline. The top-level transformation proof is simple enough to be carried out on paper, but we have mechanized enough of the theory in the Isabelle theorem prover [20] to have verified it semi-automatically, using Isabelle's powerful rewriting engine.

Interestingly, both circuits and laws can be expressed diagrammatically. A paper proof (transformation using equivalence laws) proceeds as a series of microarchitecture block diagrams, each an incrementally transformed version of the last. The laws often have a geometric flavor to them, such as laws to swap two

components with each other, or laws to absorb one component into another. We find this diagrammatic approach an excellent way to communicate proofs.

For us, the most time-consuming part of this technique has been discovering the local behavior-preserving laws. It is our experience that these laws are much easier to discover when one uses the right level of abstraction. In particular, we encapsulate all control and dataflow information concerning a given instruction in the pipeline into an abstract data type called a *transaction* [1, 17]. We have found that not only do transactions reduce the size of microarchitecture specifications, they also provide enough "auxiliary" state information to make law-discovery practical.

The rest of the paper gives a brief introduction to our specification language, and then discusses many of the laws we have discovered. We then show their use by applying the laws in a proof of equivalence between two microarchitectures. While space constraints prohibit us from giving the complete proof, the top-level proof is sketched diagrammatically in [16].

2 Specifying a Pipelined Microarchitecture

We specify microarchitectures using the *Hawk* language [4, 17]. Hawk allows us to express modern microarchitectures clearly and concisely, to simulate the microarchitectures, either directly with concrete values, or symbolically, and provides a formal basis for reasoning about their behavior at source-code level. Currently Hawk is a set of libraries built on top of the pure functional language Haskell, which is strongly typed, supports first-class functions, and infinite data structures, such as streams [8, 21]. It is this legacy that led us to look for transformation laws in the first place: one often-cited benefit of purely functional programs is that they are amenable to verification through equational reasoning. We wanted to see if such algebraic techniques scaled up to microarchitectural verification.

2.1 Hawk Signals

Hawk is a purely declarative synchronous specification language, sharing a semantic base similar to Lustre[7]. The basic data structure underlying Hawk is the *signal*, which can be thought of as an infinite sequence of values, one per clock cycle, and circuits are pure functions from input signals to output signals. The elements of a signal must belong to the same type.

We use a notion of *transactions* to specify the immediate state of an entire instruction as it travels through the microprocessor [1]. A transaction is a record with fields containing the instruction's opcode, source register names and values, and the destination register name and its value, plus any additional information, like the speculative branch target PC for each branching instruction. A microarchitecture is a network of components, each of which processes signals of transactions.

Figure 1 shows the diagram of a simple one-stage microarchitecture, built out of transaction signal processors. Each component incrementally assigns values to various transaction fields, based on the component's internal state (if any) and the values of transaction fields assigned by earlier components. A textual Hawk specification of this circuit consists of set of mutually-recursive stream equations between the components. However, in this paper we will represent Hawk circuits as diagrams.

For example, the `regFile` component has two transaction signal inputs and one transaction signal output. At a given clock cycle, the first input (called `regFileIn` in Figure 1) contains a transaction whose opcode and register name fields have been initialized, but whose value fields have all been zeroed out. The second input (called `writeback`) contains the completed transaction from the previous clock cycle. The `regFile` component first updates its internal register file state, based on the destination register name and value fields of the `writeback` input. It then fills in the source operand value fields of the `regFileIn` transaction based on the corresponding operand register names and the updated register file, and outputs the filled in transaction, all within the same clock cycle.

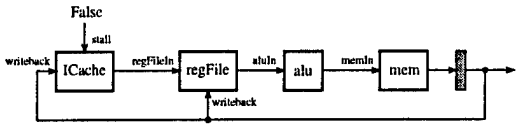


Fig. 1. One-stage pipeline.

The `alu` component examines the opcode and source operand value fields of the transaction output by `regFile`. If the opcode is an ALU operation (which include branch instructions), the `alu` component computes the appropriate result, assigns the result to the destination operand value field of the transaction, and outputs the transaction along the `memIn` wire, again within the same (long) clock cycle. If the opcode is not an ALU operation, the `alu` component outputs the transaction unchanged.

The `mem` component behaves similarly for memory load and store operations. Like the `regFile` component, the `mem` component has internal state, representing the contents of data memory at each clock cycle. This state is updated and referenced based on the transactions sent to the `mem` component. Just as with the `alu` component, all memory and transaction updating occurs within the same clock cycle. The `mem` component sends the completed transaction to a `delay` component (represented in our diagrams as a shaded box), to make it available to the `ICache` and `regFile` components in the next clock cycle. These transactions also become the output of the entire microarchitecture, as is shown by the rightmost arrow. The initial value output by the `delay` component is the default transaction `nopTrans`, which represents an "inert" transaction which behaves like a NOP instruction, but does not affect the `ICache`'s program counter.

The `ICache` component produces new transactions, based on the value of the current program counter and the contents of program memory (the instruction-set architectures we consider have separate address spaces for instructions and data). Both the current PC and the instruction memory contents are internal

to `ICache`. The ICache takes on its `writeback` input the completed transaction from the previous clock cycle. The ICache examines the transaction for branches that have been taken. When it finds such an instruction, it modifies its internal PC accordingly and starts fetching transactions from the branch target address. The ICache has as output a signal of transactions representing the newly-fetched instructions. Each transaction's source and destination operand values are initialized to zero, since the ICache doesn't know what values they should have. The other pipeline components will fill in these fields with their correct values. The ICache has a second input, called `stall`, which is a signal of Boolean values. On clock cycles where `stall` is asserted, the ICache will output the same transaction as it did on the previous clock cycle. In this simple microarchitecture, `stall` is always false. In more complex pipelines, the `stall` signal is typically asserted when the pipeline needs to stall due to a branch misprediction.

For more complex pipelines, we also allow the ICache to perform branch prediction, based on an internal branch target buffer. When performing branch prediction, the ICache will also annotate branch instruction transactions with the predicted branch target PC. A `branch_misp` component (not shown in Figure 1) can locally compare the predicted branch target with the actual branch target to determine if a branch misprediction has occurred.

3 Microarchitecture Laws

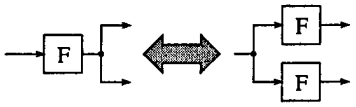


Fig. 2. Universal circuit-duplication law

With any algebraic reasoning there need to be some ground rules. We take as fundamental the notion of *referential transparency* or, in hardware terms, a *circuit duplication law.* Any circuit whose output is used in multiple places is equivalent to duplicating the circuit itself, and using each output once. This law is shown graphically in Figure 2. Because of the declarative nature of our specification language, every circuit satisfies this law. That is, it is impossible within Hawk for a specification of a component to cause hidden side-effects observable to any other component specification. In many specification languages this law does not hold universally. For example, duplicating a circuit that incremented a global variable on every clock cycle would cause the global variable to be incremented multiple times per clock period, breaking behavioral equivalence. Hawk circuits can still be stateful, but all stateful behavior must be local and/or expressed using feedback.

The next few sections introduce many other laws, some of which are specific to particular combinations of components, while others are quite widely applicable. Each instantiation of a law needs to be proved with respect to the specification of the circuit components involved. We have found induction and bisimulation to be the most useful ways of proving the laws in this paper, expressed as proofs in Isabelle.

3.1 Delay Laws

The delay circuit is a fundamental building block of clocked circuits, especially when combined with feedback. A feedback variant of the circuit duplication law shown in Figure 3, called the *feedback rotation* law, allows circuits to be split along feedback wires. This law is not universal, but it is valid for any circuit that does not contain zero-delay cycles (amongst others). Happily, all of the laws we discuss, including the feedback rotation law itself, preserve a well-formedness property: if a circuit contains no zero-delay cycles, then any transformed circuit will also have no zero-delay cycles.

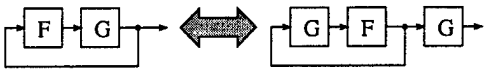


Fig. 3. feedback rotation law

The *time-invariance* law (Figure 4) is also nearly universal. A circuit is *time-invariant* if one can retime the circuit by removing the `delays` from all the inputs of the circuit and placing new `delays` on the circuit's outputs. Any combinatorial circuit that preserves default values is automatically time-invariant, but so are stateful circuits like the register file and memory cache. Interestingly, the ICache is not.

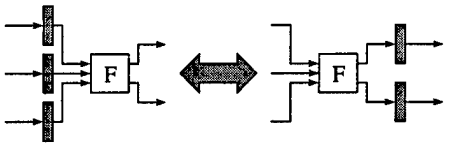


Fig. 4. time-invariance law.

We use the above laws extensively to remove pipeline stages. If a pipeline stage is time-invariant, then we can move the pipeline registers (represented as `delay` circuits) from before the pipeline stage to afterwards. If subsequent pipeline stage are also time-invariant, then we can repeat the process, eventually moving all of the delay circuits to the end of the pipeline. However, forwarding logic between pipeline stages must still access the appropriate time-delayed outputs of later pipeline stages. The feedback-rotation law polices this, and ensures that the appropriate time-delay is kept by forcing `delays` to be inserted on all feedback wires to the forwarding circuits.

3.2 Bypasses and Bypass Laws

The purpose of forwarding logic in a pipeline is to ensure that results computed in later pipeline stages are available to earlier pipeline stages in time to be used. Conceptually, the forwarding logic at each pipeline stage examines its current instruction's source operand register names to see if they match a later stage's destination operand register name. For every matching source operand, the operand value is replaced with the result value computed by the later pipeline stage. Non-matching source operands continue to use operand values given by the preceding pipeline stage.

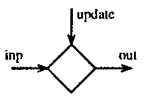


Fig. 5. bypass circuit

This conceptual logic can be implemented concisely using transactions. A *bypass* circuit (Figure 5) has two inputs, each a signal of transactions: The first input (`inp`) contains the transactions from the preceding pipeline stage. The second input (`update`) contains the transactions from a subsequent pipeline stage. The `bypass` circuit at each clock cycle compares the source operand names of the current `inp` transaction with the destination operand names of the current `update` transaction. The output of `bypass` is identical to `inp`, except that source operands matching `update`'s destination operand are updated. Bypasses arise frequently enough in pipeline specifications that we draw them specially, as diamonds with the `update` input connected to either the top or the bottom.

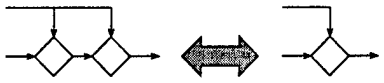


Fig. 6. bypass circuit idempotence law

Bypass circuits have many nice properties. Not only are they time-invariant and obey a kind of idempotence (Figure 6), but they also interact closely with register files and various execution units.

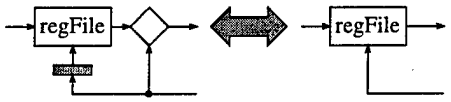


Fig. 7. register-bypass law

The fundamental interaction between a bypass and register file is shown in Figure 7. We call this the *register-bypass law*, and it is used repeatedly in eliminating forwarding logic when simplifying pipelines. The law states that we can delay writing a value into the register file, so long as we also forward the value to be written, in case that register was being read on the same clock cycle.

Initially we considered this law to be a theorem about register files, and accordingly we proved that it held for a number of different implementations. However, it is also tempting to view this law as an *axiom* of register files. In effect, by using the law repeatedly from right to left, we obtain a specification for how the register file must behave for any time prefix.

Hazard - Bypass Law Another bypass law permits the removal of bypasses between execution units. It is often the case that after retiming all delay circuits to the end of a pipeline, two execution units in a pipeline (such as an ALU unit and a Load/Store unit) are connected with one-cycle feedback loops. Each bypass circuit is forwarding the outputs of an execution unit to the inputs of that same execution unit, one clock cycle later.

If the upstream pipeline stages can guarantee that there is no hazard between successive transactions, then the double feedback is equivalent to the single feed-

back circuit shown at the bottom of Figure 8. This (conditional) identity is called the *hazard-bypass* law.

To be more concrete, suppose exec1 is the ALU and exec2 the memory cache. Then an ALU-mem hazard arises if a transaction which loads a register value from memory is immediately followed by an ALU operation which requires that register's value. Under these circumstances the two feedback loops would give different results. Under all other circumstances the two circuits are equivalent. We express this conditional equivalence using the `no_haz` component. It is an example of a projection component and is discussed in the next section.

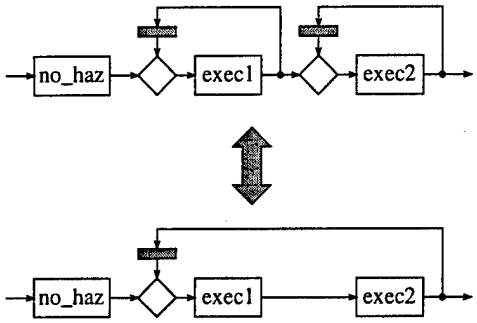


Fig. 8. hazard-bypass law

3.3 Projection Laws

Many laws, like the hazard-bypass law above, require that the input signals satisfy certain properties, and commonly, we may know that the output signal of a given component always satisfies a particular property. We can capture this knowledge of properties using signal *projections.*

A signal projection is a component with one input and one output. As long as the input signal satisfies the property of interest, the component acts like an identity function, returning the input signal unchanged. However, if the input does not satisfy the property we are interested in, the projection component modifies the input signal in some arbitrary way so that the property is satisfied.

Let us consider an example. For the hazard-bypass law we are interested in expressing the absence of ALU-mem hazards in a transaction signal. We reify this property as a `no_haz` projection. On each clock cycle, the `no_haz` component compares the current input transaction with the previous input transaction. If there is no ALU-mem hazard between the two transactions, then the current transaction is output unchanged. If a hazard does exist, then `no_haz` will instead output `nopTrans`, which is guaranteed not to generate a hazard (since `nopTrans` contains no source operands).

Where do projections come from? After all, they are not the sort of component that microarchitectural designers introduce just for fun.

Fig 9 provides an example of a law which "generates" a projection. The hazard-squashing logic guarantees that its output contains no hazards, and this is expressed in that the circuit is unchanged when the `no_haz` component is inserted on its output.

(The `hazard` component outputs a Boolean on each clock cycle stating whether its two input transactions constitute a hazard. The `kill` component takes a transaction signal and a Boolean signal as inputs. On each clock cycle, if the

Boolean input is false, then `kill` outputs its input transaction unchanged. If the Boolean input is true, then `kill` outputs a `nopTrans`, effectively "killing" the input transaction.)

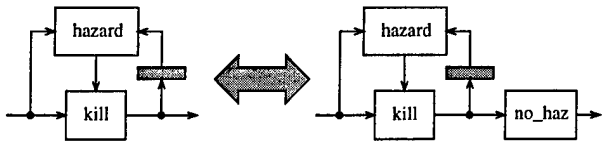


Fig. 9. Hazard-squashing logic guarantees no hazards

To be useful, a projection component needs to be able to migrate from a source circuit that produces it (such as the circuit in Figure 9) to a target circuit that needs the projection to enable an algebraic law (such as the hazard-bypass law). Thus a projection component must be able to commute with the intervening circuits between the source and the target circuit. Well-designed projections commute with many circuits. For instance, the `no_haz` projection commutes with `bypass`, `alu`, `mem`, and `regFile` components. It also commutes with `delay` components (that is, `no_haz` is time-invariant).

Projections are also convenient for expressing the fact that a component only uses some of the fields of an input transaction. For instance, the `hazard` component only looks at the opcode, source, and destination register name fields of its two input transactions. We can create a projection called `proj_ctrl` that sets every other field of a transaction to a default value, and prove a law stating that the `hazard` component is unchanged when `proj_ctrl` is added to any of its inputs. We can then show that `proj_ctrl` commutes with other components, such as bypasses and delays. This allows us to move the input wires to `hazard` across these other components, which is sometimes necessary to enable other laws. Similarly, the `proj_branch_info` projection allows us to move `ICache` and `branch_misp` component inputs.

4 Transforming the Microarchitecture

The laws we have been discussing can be used for aggressively restructuring microarchitectures while retaining equivalence. We have used them to simplify several pipelined microarchitectures with a view to verification. The example we present here contains three levels of forwarding logic, resolves hazards by stalling the pipeline, and performs branch speculation. The block diagram for this microarchitecture is shown in Figure 10.

By using just algebraic laws, we have been able to reduce most of the complexity, leaving essentially an unpipelined microarchitecture. We are currently implementing the algebraic laws as a rewrite system in Isabelle. For this paper we describe our top-level rewrite strategy informally.

Retiming We first remove all delay circuits from the main pipeline path. We accomplish this by repeatedly applying the time-invariance law, and by splitting delays along wires through the circuit duplication and feedback rotation laws.

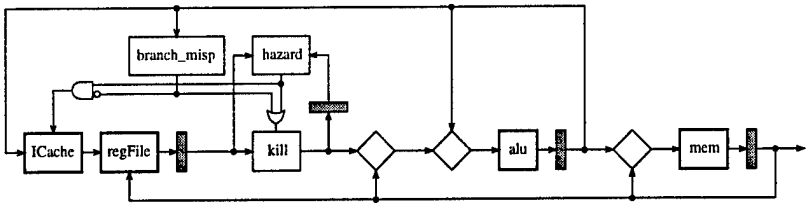


Fig. 10. Microarchitecture before simplification

Move control wires Next, we move all wires not directly involved with forwarding logic to either before or after all of the `bypass` circuits. This is to enable the hazard-bypass laws, which we apply in a later step. We move the wires by inserting projection circuits and using the corresponding projection-commutativity laws.

Propagate hazard information The hazard-bypass laws can only be applied when there are no hazards between the affected stages. So we generate a no-hazard projection at the end of the dispatch stage (which is justified by a projection-absorption law applicable to the kill-circuit complex in that stage), and then move it between the first and second bypass circuits. We also use additional properties of the `proj_ctrl`, `kill`, and `regFile` circuits (discussed in [16]) to swap the hazard/kill complex with the register file, so that the register-bypass law can be used more readily in the next step of the simplification. The circuit in Figure 11 shows the microarchitecture after this step has been completed. Notice that the ALU and memory units are now connected exactly as required for an application of the hazard-bypass law.

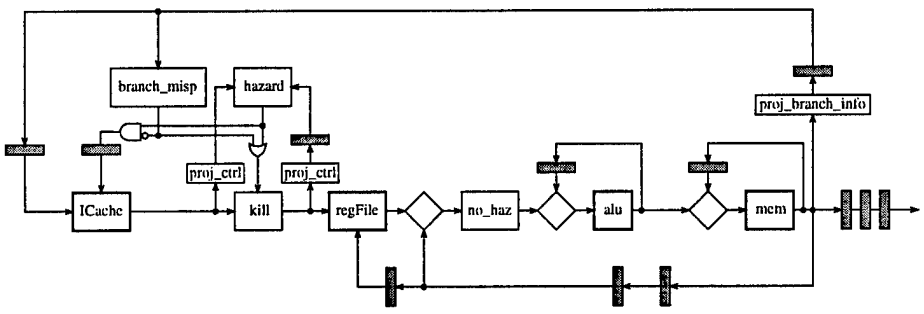


Fig. 11. Microarchitecture after the "propagate hazard information" step

Remove forwarding logic We can now apply the hazard-bypass law to remove the bypass circuit just prior to the memory unit. We eliminate the other two bypass circuits by applying the register-bypass law twice.

Cleanup The pipeline has now been simplified as much as possible, except that there are still some extra delay components as well as several unnecessary projection circuits. We merge delay components, then move the projection circuits back to their places of origin and remove them using the projection laws in the opposite direction.

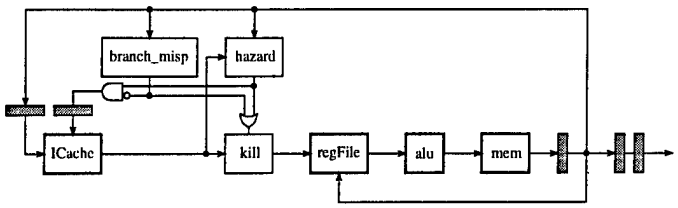


Fig. 12. Microarchitecture after simplification

The final microarchitecture is shown in Figure 12. This circuit still outputs exactly the same transaction values, cycle-for-cycle, as the microarchitecture in Figure 10, but is considerably less complex. We can now apply conventional techniques to verify that this microarchitecture is a valid implementation of the ISA.

5 Discussion

5.1 Related work

Hawk is built on top of the pure functional language Haskell, where algebraic techniques for transforming functional programs are routinely used for equivalence checking and verification [2, 3, 13] and for compilation and optimization [5, 12]. Much of our work can be seen as an extension of these ideas. Hawk itself is very similar in flavor to Lustre [6] except that in Lustre signals are accompanied by additional clock information. The Hawk specification style follows from the work of Johnson[9], O'Donnell[18], and Sheeran[25].

We have also been influenced by the algebraic techniques used in the relational hardware-description language Ruby [24]. Sizeable Ruby circuits have been successfully derived and verified through algebraic manipulation [10, 11]. What distinguishes our work is our focus on microarchitectural units as objects of study in their own right. The Ruby research has emphasized circuits at the gate level.

In terms of verification, our approach is most similar to two known techniques, called retiming [14, 23, 26] and unpipelining [15]. A circuit is *retimed*

when the delay components of the circuit are repositioned, while the functional components are left unchanged, effectively through repeated applications of the time-invariance law. Typically, circuits are retimed to reduce the clock cycle time. In contrast, we retime circuits as part of a simplification process. In fact, we often use the time invariance law to increase cycle time!

Unpipelining [15] is a verification technique where a pipelined microarchitecture, specified as a state machine, is incrementally transformed into a functionally-equivalent unpipelined microarchitecture. Unpipelining proceeds by repeatedly merging the last stage of a pipeline into the next to last stage, producing a microarchitecture with one less stage on each iteration. On each iteration, the two microarchitectures are proven equivalent by induction over time. This is similar to our approach, except that we use transactions to encapsulate and reuse many of the verification steps, and we only need to prove the equivalence of the portion of the microarchitecture being transformed, rather than the entire microarchitecture, on each iteration. On the other hand, Levitt and Olukotun's implementation of unpipelining is much more automated than our work up to now.

Transactions were a key concept in allowing us to discover and formulate many of the algebraic laws of microarchitectural components. Unsurprisingly, the usefulness of transactions has been noticed before. Aagaard and Leeser used transactions to specify and verify hierarchical networks of pipelines [1], and Önder and Gupta have used a similar concept of *instruction contexts* as a core datatype in UPFAST, an imperative microarchitecture simulation language [19]. Further, Sawada and Hunt use an extended form of transactions in their verification of a speculative out-of-order microarchitecture [22]. Each transaction records two snapshots of the entire ISA state, before and after the instruction is executed. In their work, however, transactions are not part of the microarchitecture itself, but are constructed separately for verification purposes.

5.2 Next steps in microarchitecture algebra

As we have come to see it, the main principle of applying algebraic techniques to microarchitectures is to use geometric reasoning to move and absorb circuits, and to express that reasoning as local equalities whenever possible. Conditional equalities can be expressed using projections.

Some care is required in the definition of basic components. We have striven to design the component circuits to satisfy as rich a variety of algebraic laws as possible, such as preserving default values, satisfying time-invariance, and so on. Sometimes we hit on the correct definitions immediately, but more commonly adapted the definitions over time admitting more and more laws. One example of this is in pipeline registers. Initially, we used conditional delays to act as pipeline registers, but since then have found it useful to separate clocked behavior from functional behavior, enabling the two dimensions to be manipulated separately.

In some sense the components we now manipulate are not optimal in terms of transistor counts. In particular, many units receive and propagate information they are not interested in. However, much of this overhead can be removed

automatically through a similar set of rewrite laws built around more primitive components than those presented in this paper. We plan to write this up in a subsequent paper.

6 Acknowledgements

We wish to thank Borislav Agapiev, Carl Seger, Byron Cook, Sava Krstic, and Thomas Nordin for their valuable contributions to this research. The authors are supported by Intel Strategic CAD Labs and Air Force Material Command (F19628-93-C-0069). John Matthews receives support from a graduate research fellowship with the NSF.

References

1. AAGAARD, M., AND LEESER, M. Reasoning about pipelines with structural hazards. In *Second International Conference on Theorem Provers in Circuit Design* (Bad Herrenalb, Germany, Sept. 1994).
2. BIRD, R., AND WADLER, P. *Introduction to Functional Programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1988.
3. BIRD, R. S., AND MOOR, O. D. *Algebra of Programming.* Prentice Hall, 1996.
4. COOK, B., LAUNCHBURY, J., AND MATTHEWS, J. Specifying superscalar microprocessors in Hawk. In *FTH'98, Workshop on Formal Techniques for Hardware and Hardware-like Systems* (Marstrand, Sweden, June 1998).
5. GILL, A., LAUNCHBURY, J., AND PEYTON JONES, S. L. A Short Cut to Deforestation. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark, June 1993), ACM Press, pp. 223–232.
6. HALBWACHS, N. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.
7. HALBWACHS, N., LAGNIER, F., AND RATEL, C. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems* (September 1992).
8. HUDAK, P., PETERSON, J., AND FASEL, J. A gentle introduction to Haskell. Available at `www.haskell.org`, Dec. 1997.
9. JOHNSON, S. D. *Synthesis of Digital Systems from Recursive Equations.* ACM Distinguished Dissertation Series. MIT Press, 1984.
10. JONES, G., AND SHEERAN, M. Collecting butterflies. Tech. rep., Oxford University Computing Laboratory, 1991.
11. JONES, G., AND SHEERAN, M. Designing arithmetic circuits by refinement in ruby. In *Mathematics of Program Construction* (1993), vol. 669 of *LNCS*, Springer Verlag.
12. JONES, S. L. P., AND SANTOS, A. L. M. A transformation-based optimiser for Haskell. *Science of Computer Programming 32*, 1–3 (Sept. 1998), 3–47.
13. LAUNCHBURY, J. Graph algorithms with a functional flavour. *Lecture Notes in Computer Science 925* (1995).
14. LEISERSON, C. E., AND SAXE, J. B. Retiming synchronous circuitry. *Algorithmica 6* (1991), 5–35.

15. LEVITT, J., AND OLUKOTUN, K. A scalable formal verification methodology for pipelined microprocessors. In *33rd Design Automation Conference (DAC'96)* (New York, June 1996), Association for Computing Machinery, pp. 558–563.
16. MATTHEWS, J., AND LAUNCHBURY, J. Elementary microarchitecture algebra: Top-level proof of pipelined microarchitecture. Tech. Rep. CSE-99-002, Oregon Graduate Institute, Computer Science Department, Portland, Oregon, Mar. 1999.
17. MATTHEWS, J., LAUNCHBURY, J., AND COOK, B. Specifying microprocessors in Hawk. In *IEEE International Conference on Computer Languages* (Chicago, Illinois, May 1998), pp. 90–101.
18. O'DONNELL, J. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Symposium on Functional Programming Languages in Education* (July 1995).
19. ÖNDER, S., AND GUPTA, R. Automatic generation of microarchitecture simulators. In *IEEE International Conference on Computer Languages* (Chicago, Illinois, May 1998), pp. 80–89.
20. PAULSON, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.
21. PETERSON, J., ET AL. Report on the programming language Haskell: A non-strict, purely functional language, version 1.4. Available at `www.haskell.org`, Apr. 1997.
22. SAWADA, J., AND HUNT, W. A. Processor verification with precise exceptions and speculative execution. *Lecture Notes in Computer Science 1427* (1998), 135–146.
23. SAXE, J., AND GARLAND, S. Using Transformations and Verifications in Circuit Design. *Formal Methods in System Design 4*, 1 (1994), 181–210.
24. SHARP, R., AND RASMUSSEN, O. An introduction to Ruby. Teaching Notes ID–U: 1995-80, Dept. of Computer Science, Technical University of Denmark, October 1995.
25. SHEERAN, M. *μFP, an Algebraic VLSI Design Language.* PhD thesis, Programming Research Group, University of Oxford, 1983.
26. SHEERAN, M. Retiming and slowdown in Ruby. In *The Fusion of Hardware Design and Verification* (Glasgow, Scotland, July 1988), G.J. Milne, Ed., IFIP WG 10.2, North-Holland, pp. 289–308.

Recursive Function Definition over Coinductive Types

John Matthews

Oregon Graduate Institute,
P.O. Box 91000, Portland OR 97291-1000, USA
johnm@cse.ogi.edu
http://www.cse.ogi.edu/~johnm

Abstract. Using the notions of *unique fixed point*, *converging equivalence relation*, and *contracting function*, we generalize the technique of well-founded recursion. We are able to define functions in the Isabelle theorem prover that recursively call themselves an infinite number of times. In particular, we can easily define recursive functions that operate over coinductively-defined types, such as infinite lists. Previously in Isabelle such functions could only be defined corecursively, or had to operate over types containing "extra" bottom-elements. We conclude the paper by showing that the functions for filtering and flattening infinite lists have simple recursive definitions.

1 Well-founded recursion

Rather than specify recursive functions by possibly inconsistent axioms, several higher order logic (HOL) theorem provers[4, 11, 14] provide well-founded recursive function definition packages, where new functions can be defined conservatively. Recursive functions are defined by giving a series of pattern matching reduction rules, and a well-founded relation.

For example, the *map* function applies a function f pointwise to each element of a finite list. This function can be defined using well-founded recursion:

$$map :: (\alpha \to \beta) \to \alpha\ list \to \beta\ list$$

$$\begin{aligned} map\ f\ [] &= [] \\ map\ f\ (x\#xs) &= (f\ x)\#(map\ f\ xs) \end{aligned}$$

The first rule states that *map* applied to the empty list, denoted by [], is equal to the empty list. The second rule states that *map* applied to a list constructed out of the head element x and tail list xs, denoted by $x\#xs$, is equal to the list formed by applying f to x and *map* f to xs recursively.

To define a function using well-founded recursion, the user must also supply a *well-founded relation* on one of the function's arguments[1]. A well-founded relation $(<)$ is a relation with the property that there exists no infinite sequence of elements $x_1, x_2, x_3, x_4, \ldots$ such that

$$\ldots < x_4 < x_3 < x_2 < x_1$$

For each reduction rule, the recursive definition package checks that every recursive call on the right-hand side of the rule is applied to a smaller argument than on the left-hand side, according to the user supplied well-founded relation.

[1] Some well-founded recursion packages only allow single-argument functions to be defined. In this case one can gain the effect of multi-argument curried functions by tupling.

In the case of *map*, we can supply the well-founded relation

$$xs < ys \equiv length\ xs < length\ ys$$

which is true when the number of elements in the relation's left-hand list argument is less than the number of elements in the relation's right-hand argument. The definition of *map* contains only one recursive rule, and it is easy to prove that the xs argument of the recursive call of *map* is smaller than the $(x\#xs)$ argument on the left-hand side of the rule, according to this relation. In general, well-founded relations ensure that there are no infinite chains of nested recursive calls.

2 Coinductive types and corecursive functions

Although well-founded recursion is a useful definition technique, there are many recursive definitions that fall outside its scope. For instance, there is a non-inductive type of *lazy lists* in the Isabelle[11] theorem prover, denoted by $\alpha\ llist$, that is the set of all finite and infinite lists of type α. The function *lmap* over this type is uniquely specified by the following recursive equations[2]:

$$\begin{aligned} lmap\ f\ [] &= [] \\ lmap\ f\ (x\#xs) &= (f\ x)\ \#\ (lmap\ f\ xs) \end{aligned}$$

One cannot define *lmap* using well-founded recursion since the length of an infinite list does not decrease when you take its tail. In fact, the expression
$lmap\ f\ (x_1\ \#\ x_2\ \#\ x_3\ \#\ \ldots)$ can be unfolded using the above rules to an infinite chain of recursive calls:

$$\begin{aligned} & lmap\ f\ (x_1\ \#\ x_2\ \#\ x_3\ \#\ ...) \\ = \\ & (f\ x_1)\ \#\ (lmap\ f\ (x_2\ \#\ x_3\ \#\ ...)) \\ = \\ & (f\ x_1)\ \#\ (f\ x_2)\ \#\ (lmap\ f\ (x_3\ \#\ ...)) \\ = \\ & (f\ x_1)\ \#\ (f\ x_2)\ \#\ (f\ x_3)\ \#\ (lmap\ f\ (...)) \\ = \\ & ... \end{aligned}$$

Defining functions corecursively

The $\alpha\ llist$ type is an example of a coinductive type. Although there is no general induction principle for coinductive types, one can use principles of coinduction to show that two coinductive values are equal, and one can build coinductive values using *corecursion.*

In Isabelle's theory of lazy lists[12], for instance, one builds potentially infinite lists through the *llist_corec* operator, which has type $\beta \to (\beta \to unit + (\alpha * \beta)) \to (\alpha\ llist)$. The *llist_corec* operator uniquely satisfies the following recursion equation:

$$llist\_corec\ b\ g = \begin{cases} [], & \text{if } g\ b = \mathrm{Inl}\ () \\ (x\ \#\ (llist\_corec\ b'\ g)), & \text{if } g\ b = \mathrm{Inr}\ (x, b') \end{cases}$$

The *llist_corec* operator takes as arguments an initial value b and a function g. When g is applied to b, it either returns Inl (), indicating that the result list should be empty,

[2] Isabelle uses a different syntax for lazy lists than for finite lists. In this paper we use the same syntax for both types.

or the value $\mathrm{Inr}\,(x, b')$, where x represents the first element of the result list, and b' represents the new initial value to build the rest of the list from. Function g is called iteratively in this fashion, constructing a potentially infinite list.

Using *llist_corec*, we can define *lmap* corecursively as follows:

$$
\begin{array}{l}
\mathit{lmap}\ f\ \mathit{xs} \equiv \mathit{llist\_corec}\ \mathit{xs}\ (\mathit{map\_head}\ f) \\
\text{where} \\
\quad \mathit{map\_head}\ :: (\alpha \to \beta) \to \alpha\ \mathit{llist} \to (\mathit{unit} + (\beta * \alpha\ \mathit{llist})) \\
\quad \mathit{map\_head}\ f\ \mathit{xs} \equiv \text{case}\ \mathit{xs}\ \text{of} \\
\qquad\qquad\qquad [] \Rightarrow \mathrm{Inl}\,() \\
\qquad\qquad \mid\ (x \# \mathit{xs}') \Rightarrow \mathrm{Inr}\,(f\,x, \mathit{xs}')
\end{array}
$$

One can then prove by coinduction that this definition satisfies *lmap*'s recursive equations. Needless to say, this is not the most intuitive specification of *lmap*, and most people would prefer to specify such functions using recursion, if possible. In the remainder of the paper we will present a framework for defining functions such as *lmap* recursively.

3 Solving recursive equations

The basic steps required in this framework to show that a set of recursive equations is well defined are as follows:

- Construct a single function F that characterizes the set of recursive equations.
- Show that for any two different potential solutions supplied to F, F maps them to two potential solutions that are closer together, in a suitable sense.
- Invoke the main result (Sect. 4.3) to show that the above property of F is sufficient to guarantee that there is a unique solution to the original set of recursive equations.

In this section we deal with the first step.

3.1 Unique fixed points

We convert a system of pattern matching recursive equations into a functional form by employing a standard technique from domain theory[5, 17]. We start by recasting the equations as a single recursive equation using argument destructors or nested case-expressions. For example, the recursive equations defining the *lmap* function are equivalent to the following single recursive equation:

$$
\begin{array}{l}
\mathit{lmap}\ f\ l = \text{case}\ l\ \text{of} \\
\qquad\qquad [] \Rightarrow [] \\
\qquad \mid\ (x \# \mathit{xs}) \Rightarrow (f\,x) \# (\mathit{lmap}\ f\ \mathit{xs})
\end{array}
$$

Given f, we can reify this pattern of recursion into a non-recursive function F of type $(\alpha\ \mathit{llist} \to \beta\ \mathit{llist}) \to (\alpha\ \mathit{llist} \to \beta\ \mathit{llist})$ that takes a functional parameter *lmap_f*:

$$
\begin{array}{l}
F\ \mathit{lmap\_f} = \lambda l\,.\ \text{case}\ l\ \text{of} \\
\qquad\qquad [] \Rightarrow [] \\
\qquad \mid\ (x \# \mathit{xs}) \Rightarrow (f\,x) \# (\mathit{lmap\_f}\ \mathit{xs}).
\end{array}
$$

Using the recursive equations for *lmap*, it is easy to show that $\mathit{lmap}\ f = F\,(\mathit{lmap}\ f)$. The value $\mathit{lmap}\ f$ is called a *fixed point* of F. In general, an element x of type α is a fixed point of a function g of type $\alpha \to \alpha$ if $x = g\,x$. A function may have many fixed points, or none at all. Considering g as a functional representation of a system of recursive equations, each fixed point of g represents a valid solution to the system. If the function g has exactly one fixed point x, then we can think of g as *defining* the value x, in a way that will be made precise shortly.

Definition 1 *A function f of type $\alpha \to \alpha$ has a* unique fixed point *element x of type α if $x = f\,x$ and*

$$\forall y\, z\,.\,(y = f\,y) \wedge (z = f\,z) \longrightarrow y = z.$$

We formalize this definition into a predicate of higher order logic:

$$\begin{aligned} &\mathit{isUniqFix} :: \alpha \to (\alpha \to \alpha) \to \mathit{bool} \\ &\mathit{isUniqFix}\,x\,f \equiv \quad x = f\,x \wedge (\forall y\, z\,.\,f\,y = f\,z \longrightarrow y = z) \end{aligned}$$

To define elements using unique fixed points, we rely on Hilbert's description operator (ε):

$$\begin{aligned} &\mathrm{fix} :: (\alpha \to \alpha) \to \alpha \\ &\mathrm{fix}\,f \equiv \varepsilon x\,.\,\mathit{isUniqFix}\,x\,f \end{aligned}$$

The expression $\mathrm{fix}\,f$ represents the unique fixed point of f, when one exists. The following lemma captures this fact:

Lemma 1 *If there exists an x such that $\mathit{isUniqFix}\,x\,f$ holds, then*

$$x = \mathrm{fix}\,f = f\,(\mathrm{fix}\,f)$$

If f does not have a *unique* fixed point, then $\mathrm{fix}\,f$ denotes an arbitrary value.

3.2 Properties of unique fixed points

As an aside, several nice properties hold when one can establish that a system of recursive equations has a unique solution. For example, unique fixed points can sometimes "absorb" functions applied to other fixed points.

Lemma 2 *Given functions $F : \alpha \to \alpha$, $G : \beta \to \beta$, $f : \alpha \to \beta$, and value $x : \alpha$, such that x is a (not necessarily unique) fixed point of F, G has a unique fixed point, and $f \circ F = G \circ f$, then $f\,x = \mathrm{fix}\,G$.*

Proof We have $f\,x = f\,(F\,x) = (f \circ F)\,x = (G \circ f)\,x = G\,(f\,x)$. Thus the value $f\,x$ is a fixed point of G. But since $\mathrm{fix}\,G$ is the unique fixed point of G, then $f\,x = \mathrm{fix}\,G$ **qed**

Unique fixed points can also be "rotated", in the following sense:

Lemma 3 *If the composition of two functions $g : \beta \to \alpha$ and $h : \alpha \to \beta$ has a unique fixed point $\mathrm{fix}\,(g \circ h)$, then $h \circ g$ also has a unique fixed point, and $\mathrm{fix}\,(g \circ h) = g\,(\mathrm{fix}\,(h \circ g))$.*

Proof Let $b = \mathrm{fix}\,(g \circ h)$. We first note that $h\,b = h\,((g \circ h)\,b) = (h \circ g)\,(h\,b)$. Thus $h\,b$ is a fixed point of $h \circ g$. Next we show that this fixed point is unique by showing that any two fixed points of $h \circ g$ are equal.

Suppose x and y are fixed points of $h \circ g$. Then $g\,x = g\,((h \circ g)\,x) = (g \circ h)\,(g\,x)$. Thus $g\,x$ is a fixed point of $g \circ h$. But since $g \circ h$ has a unique fixed point, then $g\,x = \mathrm{fix}\,(g \circ h)$. Similarly, $g\,y = \mathrm{fix}\,(g \circ h)$, and so $g\,x = g\,y$. Applying h to both sides of this equality, we obtain $h\,(g\,x) = h\,(g\,y)$, which is the same as $(h \circ g)\,x = (h \circ g)\,y$. Since both x and y are fixed points of $h \circ g$, we have $x = y$.

We can now apply Lemma 2, setting $F = h \circ g$, $G = g \circ h$, $f = g$, and $x = \mathrm{fix}\,(h \circ g)$, to conclude that $g\,(\mathrm{fix}\,(h \circ g)) = \mathrm{fix}\,(g \circ h)$ **qed**

Although we will not use Lemma 2 or Lemma 3 in the remainder of the paper, lemmas such as these are useful for manipulating systems of recursive equations as objects in their own right.

4 Converging equivalence relations and contracting functions

While unique fixed points are a useful definition mechanism, it can be difficult to show that they exist for a given function. A direct proof usually involves constructing an explicit fixed point witness using other definition techniques, such as corecursion or well-founded recursion. Little effort seems to be saved.

We propose an alternative proof technique, based on concepts from domain theory[5, 17] and topology[1, 13] where one builds a collection of ever-closer approximations to the desired fixed point, and show that the limit of these approximations exists, is a fixed point of the function under consideration, and is unique. The approximation process can be parameterized to some extent, and reused across multiple definitions that are "similar" enough. Furthermore these parameterized approximations can be composed hierarchically, yielding more powerful approximation techniques.

4.1 Converging equivalence relations

To make the notion of approximation precise, we need a way of stating how "close" two potential approximations are to each other. One approach would be to define a suitable metric space[1] and use the corresponding distance function, which returns either a rational or real number, given any two elements in the domain of the metric space. However, proving that a series of approximations converges to a limit point often requires one to reason about exponentiation and division over a theory of rationals or reals. An alternative way to measure "closeness", which we call a *converging equivalence relation* (CER), instead only involves reasoning about well-founded sets, such as the set of natural numbers, or the set of finite lists. In many cases we can prove a unique fixed point exists by performing a simple induction over the natural numbers, something which all of the current HOL theorem provers support well.

A converging equivalence relation consists of:

- A type α, called the *resolution space*
- A type β, called the *target space*
- A well-founded, transitive relation $(<)$ over type α, called a *resolution ordering*
- A three-argument predicate $(\approx)$ of type $(\alpha \to \beta \to \beta \to \mathit{bool})$, called an *indexed equivalence relation.* Given an element i of type α, and two elements x and y of type β, we denote the application of $(\approx)$ to i, x and y as $(x \overset{i}{\approx} y)$, and if this value is true, then we say that x and y are *equivalent at resolution* i.

The resolution ordering $(<)$ and indexed equivalence relation $(\approx)$ must satisfy the properties in Fig. 1, for arbitrary $i, i' : \alpha$; $x, y, z : \beta$; and $f : \alpha \to \beta$. Axioms (1), (2), and (3) state that $(\approx)$ must be an equivalence relation at each resolution i. Axiom (4) states that if a resolution i has no lower resolutions, then $(\approx)$ treats all target elements as equivalent at that resolution. Such resolutions are called *minimal.* There is always at least one minimal resolution (and perhaps more than one), since $(<)$ is well-founded. Axiom (5) states that if two elements are equivalent at a particular resolution, then they are equivalent at all lower resolutions. Thus higher resolutions impose finer-grained, but compatible, partitions of the target space than lower resolutions do. Although no particular resolution may distinguish all elements, (6) states that if two elements are equivalent at all resolutions, then they are in fact equal.

Axioms (7) and (8) deal with "limits" of approximations. First some terminology: a function $f : \alpha \to \beta$ from the space of resolutions to the target space of elements is called an *approximation map.* An approximation map f is *convergent up to resolution* i if for all resolutions j and j' such that $j < j' < i$, then $(f\, j)$ is equivalent at resolution j to $(f\, j')$. Note that it is possible for $(f\, i)$ itself not to be equivalent to

$$x \overset{i}{\approx} x \tag{1}$$

$$x \overset{i}{\approx} y \longrightarrow y \overset{i}{\approx} x \tag{2}$$

$$x \overset{i}{\approx} y \wedge y \overset{i}{\approx} z \longrightarrow x \overset{i}{\approx} z \tag{3}$$

$$(\forall j \,.\, \neg(j < i)) \longrightarrow x \overset{i}{\approx} y \tag{4}$$

$$x \overset{i'}{\approx} y \wedge i < i' \longrightarrow x \overset{i}{\approx} y \tag{5}$$

$$(\forall j \,.\, x \overset{j}{\approx} y) \longrightarrow x = y \tag{6}$$

$$(\forall j, j' \,.\, j < j' < i \longrightarrow (f\, j) \overset{j}{\approx} (f\, j')) \longrightarrow (\exists z \,.\, \forall j < i \,.\, z \overset{j}{\approx} (f\, j)) \tag{7}$$

$$(\forall j, j' \,.\, j < j' \longrightarrow (f\, j) \overset{j}{\approx} (f\, j')) \longrightarrow (\exists z \,.\, \forall j \,.\, z \overset{j}{\approx} (f\, j)) \tag{8}$$

Fig. 1. The CER axioms. Each of these axioms must hold for arbitrary i, x, y, and f.

any of the lower-resolution $(f\, j)$'s. An approximation map f is *globally convergent* if for all resolutions j and j' such that $j < j'$, then $(f\, j) \overset{j}{\approx} (f\, j')$.

Axiom (7) states that if f is locally convergent up to resolution i, then there exists a limit-like element z that is equivalent at each resolution $j < i$ to the corresponding $(f\, j)$ approximation. Axiom (8) states that if f is globally convergent, then there exists a limit element z that is equivalent to each approximation $(f\, j)$ at resolution j.

4.2 Examples of converging equivalence relations

Discrete CER The simplest useful CER has as a resolution space a two-element type containing the values $\bot$ and $\top$, with $(\bot < \top)$, and a target space β with $(\approx)$ defined such that $(x \overset{\bot}{\approx} y) \equiv \mathit{True}$, and $(x \overset{\top}{\approx} y) \equiv (x = y)$. Axioms (1) through (6) are easy to verify. Axiom (7) holds for any element. The limit element satisfying (8) is $f\, \top$.

Lazy list CER We can construct a converging equivalence equation for comparing coinductive lists by comparing the first i elements of two lazy lists l_1 and l_2 at a given resolution i. To perform the comparison, we make use of the *ltake* function, with type $\mathit{nat} \to \alpha\ \mathit{llist} \to \alpha\ \mathit{list}$. The expression $(\mathit{ltake}\, n\, xs)$ returns a finite list consisting of the first n elements of xs. If xs has fewer than n elements, then *ltake* returns the whole of xs. The *ltake* function can be defined by well-founded recursion on its numeric argument with the following recursive equations:

$$\begin{array}{lllll} \mathit{ltake} & 0 & xs & = & [] \\ \mathit{ltake}\ (n+1) & [] & = & [] & \\ \mathit{ltake}\ (n+1)\ (x \# xs) & = & x \# (\mathit{ltake}\, n\, xs) & & \end{array}$$

We then define the lazy list CER with the natural numbers as the resolution space, $(\alpha\ \mathit{llist})$ as the target space, the usual ordering on the natural numbers for $(<)$, and $(\approx)$ defined as follows:

$$xs \overset{i}{\approx} ys \equiv (\mathit{ltake}\, i\, xs = \mathit{ltake}\, i\, ys).$$

Axioms (1) through (3) hold trivially. The only minimal resolution in this CER is 0, and since $(\mathit{ltake}\, 0\, xs) = []$, then (4) holds. If two lazy lists are equal up to the first i positions, then they are equal up to any $i' < i$ position, so (5) holds. Axiom (6) reduces to the Take Lemma[12], which can be proved by coinduction.

Axioms (7) and (8) require us to construct appropriate limit elements, given an approximation map. Both limit elements can be constructed by a single function, which we call *llist_diag*. For a given approximation map f, the limit elements may be of infinite length, so we define *llist_diag* by corecursion, using *llist_corec*:

$$\begin{array}{l} \mathit{llist\_diag}\ f \equiv \mathit{llist\_corec}\ 0\ (\mathit{nthElem}\ f) \\ \textit{where} \\ \qquad \mathit{nthElem}\ f\ n \equiv \begin{cases} \mathrm{Inl}\,(), & \text{if } \mathit{ldrop}\ n\ (f(n+1)) = [] \\ \mathrm{Inr}\,(x, n+1), & \text{if } \mathit{ldrop}\ n\ (f(n+1)) = (x \# xs) \end{cases} \end{array}$$

The helper function *nthElem* uses the *ldrop* function on lazy lists. The *ldrop* function has type $\mathit{nat} \to (\alpha\ \mathit{llist}) \to (\alpha\ \mathit{llist})$, and $(\mathit{ldrop}\ i\ xs)$ removes the first i elements from xs, returning the remainder. Like *ltake*, it is defined by well-founded recursion on its numeric argument:

$$\begin{array}{lllll} \mathit{ldrop} & 0 & xs & = & xs \\ \mathit{ldrop} & (n+1) & [] & = & [] \\ \mathit{ldrop} & (n+1) & (x \# xs) & = & \mathit{ldrop}\ n\ xs \end{array}$$

The overall action of *llist_diag* is to construct a so-called *diagonal list* from the approximation map f, where the n^{th} element of the result list is drawn from the n^{th} element of approximation $f\,(n+1)$, if the n^{th} element exists. If the n^{th} element does not exist (i.e., the length of $f\,(n+1)$ is less than n), then the result list is terminated at that point. This process is shown in Fig. 2. There are two possible cases. In Fig. 2-a, we see that the approximation map f converges to the finite list $[x_0,\ x_1,\ x_2,\ x_3,\ x_4]$. In Fig. 2-b, the approximation map f is converging to the infinite list $[x_0,\ x_1,\ x_2,\ x_3,\ x_4,\ x_5,\ x_6,\ \ldots]$

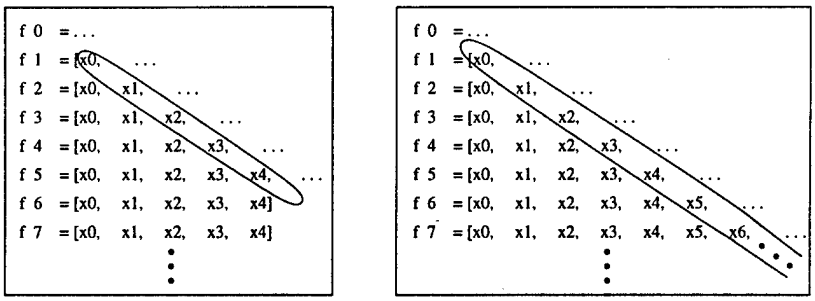


Fig. 2. The *llist_diag* function constructs a limit list from an approximation mapping. In (a) the approximation mapping converges to a finite list; In (b) to an infinite list.

It turns out that for any CER whose $(<)$ relation is the less-than ordering on the natural numbers, the following property implies both (7) and (8):

$$\forall f\,.\,(\forall i\,.\,(f\,i) \overset{i}{\approx} (f\,(i+1))) \longrightarrow (\exists x\,.\,\forall i\,.\,x \overset{i}{\approx} (f\,i)).$$

With some work, one can show that this property holds for the lazy list CER by supplying $\mathit{llist\_diag}\ f$ as the existential witness element for x.

4.3 Contracting functions

In the theory of metric spaces, a *contracting function* is a function F such that for any two points x and y, $F\,x$ is closer to $F\,y$ than x is to y, given a suitable distance function. Banach's theorem states that all contracting functions over suitable metric spaces have unique fixed points. We can define an analogous notion over a CER:

Definition 2 *A function F is* contracting *over a CER given by $(<)$ and $(\approx)$ if for all resolutions i and target elements x and y,*

$$(\forall i' < i \,.\, x \overset{i'}{\approx} y) \longrightarrow (F\,x) \overset{i}{\approx} (F\,y).$$

Intuitively a function is contracting if, given two elements x and y that are close enough together at all lower resolutions $i' < i$ to satisfy the CER, but are potentially too far away at resolution i, then F maps them to two elements that are now close enough at resolution i.

For example, the function $consZero\,xs \equiv (0\#xs)$ is contracting over the lazy list CER, since given any i and two lazy lists xs and ys,

$$(\forall i' < i \,.\, ltake\,i'\,xs = ltake\,i'\,ys) \longrightarrow ltake\,i\,(consZero\,xs) = ltake\,i\,(consZero\,ys).$$

The main result of this paper is as follows:

Theorem *A contracting function F over a CER has a unique fixed point.*

The proof is discussed in Sect. 7. For now, we would like to apply this theorem to define some simple recursive functions over lazy lists.

4.4 Recursive definitions over coinductive lists

To begin with, we can simplify the definition of a contracting function F over a CER when the $(<)$ relation of that CER is the less-than relation over the natural numbers. In this case, Definition 2 reduces to

$$\forall i\,x\,y \,.\, x \overset{i}{\approx} y \longrightarrow (F\,x) \overset{i+1}{\approx} (F\,y). \tag{9}$$

Specializing this formula for the lazy list CER, we have that F is contracting on lazy lists if

$$\forall i\,x\,y \,.\, ltake\,i\,x = ltake\,i\,y \longrightarrow ltake\,(i+1)\,(F\,x) = ltake\,(i+1)\,(F\,y). \tag{10}$$

Defining iterates Let us establish that the following recursive equation, defined over x and f, has a unique solution, and thus a definition:

$$iterates = (x \,\#\, (lmap\,f\,iterates)) \tag{11}$$

This equation builds the infinite list $[x, f\,x, f\,(f\,x), \ldots]$. We first define the non-recursive function F that characterizes this equation:

$$F\,iterates' \equiv (x \,\#\, (lmap\,f\,iterates')).$$

and then show that it is a contracting function. To do this we rely on (10), and assume we have two arbitrary lazy lists xs and ys such that $ltake\,i\,xs = ltake\,i\,ys$. We now need to show that $ltake\,(i+1)\,(F\,xs) = ltake\,(i+1)\,(F\,ys)$. Using a process of equational simplification we are able to reduce the goal to the assumption, as follows:

$$\begin{aligned}
& ltake\,(i+1)\,(F\,xs) = ltake\,(i+1)\,(F\,ys) \\
\Leftrightarrow\ & ltake\,(i+1)\,(x \,\#\, (lmap\,f\,xs)) = ltake\,(i+1)\,(x \,\#\, (lmap\,f\,ys)) \\
\Leftrightarrow\ & ltake\,i\,(lmap\,f\,xs) = ltake\,i\,(lmap\,f\,ys) \\
\Leftarrow\ & ltake\,i\,xs = ltake\,i\,ys
\end{aligned}$$

The simplification relies on the following facts, each proved by induction on i:

$$(ltake\,(i+1)\,(z\,\#\,xs) = ltake\,(i+1)\,(z\,\#\,ys)) \Leftrightarrow (ltake\,i\,xs) = ltake\,i\,ys)$$
$$(ltake\,i\,(lmap\,f\,xs) = ltake\,i\,(lmap\,f\,ys) \Leftarrow (ltake\,i\,xs = ltake\,i\,ys)$$

These facts illustrate a nice property of this proof: We did not have to expand the definitions of $(\#)$ or *lmap* during the simplification process, relying instead on an abstract characterization of their behavior with respect to *ltake*. This turns out to be the case for many functions, even recursive ones defined by contracting functions. In general we can often incrementally define recursive functions and prove properties about how they behave with respect to $(\approx)$, without having to expand the definitions of functions making up the body of the recursive definition.

5 Composing converging equivalence relations

The lazy list CER allows us to give recursive definitions of individual lazy lists, but we are often more interested in recursively defining functions that transform lazy lists. Fortunately, there are several *CER combinators* that allow us to build CERs over complex types, if we have CERs that that operate on the corresponding atomic types.

Local and global limits When constructing a new CER C' out of an existing CER C, we usually have to show (7) and (8) hold for C' by invoking (7) and (8) for C, to create the necessary limit witness elements. To make this process explicit, we use Hilbert's description operator (ε) to create functions that return these witness elements, given an appropriate approximation mapping f:

$$local\_limit :: (\alpha \to \beta) \to \alpha \to \beta$$
$$local\_limit\,f\,i \equiv (\varepsilon z\,.\,\forall j < i\,.\,z \overset{j}{\approx} (f\,j)) \tag{12}$$

$$global\_limit :: (\alpha \to \beta) \to \beta$$
$$global\_limit\,f \equiv (\varepsilon z\,.\,\forall j\,.\,z \overset{j}{\approx} (f\,j)) \tag{13}$$

We can use (7) and (8) to prove the basic properties we want *local_limit* and *global_limit* to have for any CER given by $(<)$ and $(\approx)$:

$$(\forall j, j'\,.\,j < j' < i \longrightarrow (f\,j) \overset{j}{\approx} (f\,j')) \longrightarrow (\forall j < i\,.\,(local\_limit\,f\,i) \overset{j}{\approx} (f\,j))$$
$$(\forall j, j'\,.\,j < j' \longrightarrow (f\,j) \overset{j}{\approx} (f\,j')) \longrightarrow (\forall j\,.\,(global\_limit\,f) \overset{j}{\approx} (f\,j))$$

Function-space CER The functions *local_limit* and *global_limit* allow us to concisely specify the limit elements of CER combinators. For example, given a CER C from resolution space α to target space β given by $(<)$ and $(\approx)$, we can construct a new *function-space over C* CER with the same resolution ordering $(<)$, and a new indexed equivalence relation $(\approx')$ with type
$\alpha \to (\tau \to \beta) \to (\tau \to \beta) \to bool$, defined as

$$g \overset{i}{\approx'} h \equiv \forall x\,.\,(g\,x) \overset{i}{\approx} (h\,x).$$

The limit elements satisfying (7) and (8) can be given as

$$local\_limit\_fun\,f\,i \equiv (\lambda x\,.\,local\_limit\,(\lambda i\,.\,f\,i\,x)\,i)$$
$$global\_limit\_fun\,f \equiv (\lambda x\,.\,global\_limit\,(\lambda i\,.\,f\,i\,x))$$

Given these limit-producing functions, is relatively easy to show that the function-space over C CER satisfies the CER axioms.

5.1 Defining recursive functions with the function-space CER

Defining lmap We can apply the function-space CER to define *lmap* recursively. The recursion equations for *lmap* are:

$$\begin{aligned} lmap\ f\ \ [] &= [] \\ lmap\ f\ (x\#xs) &= (f\,x)\,\#\,(lmap\ f\ xs) \end{aligned}$$

We translate the equations into a non-recursive form (parameterized over f)

$$\begin{aligned} F\ lmap' \equiv (\lambda xs.\ \text{case}\ xs\ \text{of} \\ \quad [] \quad\quad\quad &\Rightarrow [] \\ \mid (y\ \#\ ys) &\Rightarrow (f\,y)\,\#\,(lmap'\ ys)). \end{aligned}$$

We then need to show that $\text{fix}\,F$ is the unique fixed point of F by proving that F is a contracting function on the function-space over lazy lists CER. By (9) we must show for arbitrary resolution i and functions g and h, that $(g \overset{i}{\approx}' h \longrightarrow (F\,g) \overset{(i+1)}{\approx}' (F\,h))$. Expanding definitions, we obtain

$$\begin{aligned} & g \overset{i}{\approx}' h \longrightarrow (F\,g) \overset{(i+1)}{\approx}' (F\,h) \\ \Leftrightarrow\ & (\forall\, xs\,.\, g\,xs \overset{i}{\approx} h\,xs) \longrightarrow (\forall\, xs\,.\, (F\,g\,xs) \overset{(i+1)}{\approx} (F\,h\,xs)) \\ \Leftrightarrow\ & (\forall\, xs\,.\, ltake\,i\,(g\,xs) = ltake\,i\,(h\,xs)) \longrightarrow \\ & \quad (\forall\, xs\,.\, ltake\,(i+1)\,(F\,g\,xs) = ltake\,(i+1)\,(F\,h\,xs)). \end{aligned}$$

So, to prove F is contracting we take an arbitrary resolution i and two arbitrarily chosen functions g and h such that $(\forall\, xs\,.\, ltake\,i\,(g\,xs) = ltake\,i\,(h\,xs))$, and show for an arbitrary xs that $ltake\,(i+1)\,(F\,g\,xs) = ltake\,(i+1)\,(F\,h\,xs)$. There are two cases to consider:

case $xs = []$:

$$\begin{aligned} & ltake\,(i+1)\,(F\,g\,xs) = ltake\,(i+1)\,(F\,h\,xs) \\ \Leftrightarrow\ & ltake\,(i+1)\,(F\,g\,[]) = ltake\,(i+1)\,(F\,h\,[]) \\ \Leftrightarrow\ & ltake\,(i+1)\,(\text{case}\ []\ \text{of} \\ & \qquad\qquad [] \quad\quad\quad \Rightarrow [] \\ & \qquad\qquad \mid\ (y\ \#\ ys) \Rightarrow (f\,y)\,\#\,(g\,ys)) = \\ & ltake\,(i+1)\,(\text{case}\ []\ \text{of} \\ & \qquad\qquad [] \quad\quad\quad \Rightarrow [] \\ & \qquad\qquad \mid\ (y\ \#\ ys) \Rightarrow (f\,y)\,\#\,(h\,ys)) \\ \Leftrightarrow\ & ltake\,(i+1)\,[] = ltake\,(i+1)\,[] \\ \Leftrightarrow\ & \text{True}. \end{aligned}$$

case $xs = (y\#ys)$:

$$\begin{aligned} & ltake\,(i+1)\,(F\,g\,xs) = ltake\,(i+1)\,(F\,h\,xs) \\ \Leftrightarrow\ & ltake\,(i+1)\,(F\,g\,(y\#ys)) = ltake\,(i+1)\,(F\,h\,(y\#ys)) \\ \Leftrightarrow\ & ltake\,(i+1)\,(\text{case}\ (y\#ys)\ \text{of} \\ & \qquad\qquad [] \quad\quad\quad \Rightarrow [] \\ & \qquad\qquad \mid\ (y\ \#\ ys) \Rightarrow (f\,y)\,\#\,(g\,ys)) = \end{aligned}$$

$$
\begin{array}{ll}
 & \mathit{ltake}\,(i+1)\,(\text{case}\,(y\#ys)\,\text{of} \\
 & \qquad\qquad [] \Rightarrow [] \\
 & \qquad\quad \mid (y\,\#\,ys) \Rightarrow (f\,y)\,\#\,(h\,ys)) \\
\Leftrightarrow & \mathit{ltake}\,(i+1)\,((f\,y)\,\#\,(g\,ys)) = \mathit{ltake}\,(i+1)\,((f\,y)\,\#\,(h\,ys)) \\
\Leftrightarrow & \mathit{ltake}\,i\,(g\,ys) = \mathit{ltake}\,i\,(h\,ys) \\
\Leftrightarrow & \text{True}\ \{\text{by assumption}\}.
\end{array}
$$

Given the definition of F and basic lemmas about *ltake*, Isabelle's high-level simplification tactics allow the above proof to be carried out in two steps. The proof completes in about a second on a 266MHz Pentium II.

Defining lappend We can apply the function-space CER combinator repeatedly, to prove that multi-argument curried functions have unique fixed points. As a concrete example, the curried function *lappend* has type $\alpha\,\mathit{llist} \to \alpha\,\mathit{llist} \to \alpha\,\mathit{llist}$. It takes two lazy list arguments xs and ys and returns a new list consisting of the elements of xs followed by the elements of ys. The recursive equations for *lappend* are

$$
\begin{array}{ll}
\mathit{lappend} & [] \quad ys = ys \\
\mathit{lappend} & (x\#xs)\ ys = (x\,\#\,\mathit{lappend}\ xs\ ys)
\end{array}
$$

To prove that these equations have a unique solution, we apply the function-space CER combinator to the lazy list CER to obtain a new CER C'. We then apply the function-space CER combinator again to C', obtaining a new CER C'' with the usual less-than relation on *nat* for $(<)$ and the following indexed equivalence relation $(\approx'')$:

$$
g \overset{i}{\approx''} h \equiv (\forall\, xs\ ys\,.\,\mathit{ltake}\,i\,(g\,xs\,ys) = \mathit{ltake}\,i\,(h\,xs\,ys)).
$$

Next, we convert the recursive equations for *lappend* into a non-recursive function F:

$$
\begin{array}{ll}
F\ \mathit{lappend}' \equiv (\lambda xs\ ys\,.\ & \text{case}\,xs\,\text{of} \\
 & \qquad\quad [] \Rightarrow ys \\
 & \qquad \mid (x\,\#\,xs') \Rightarrow (x\,\#\,(\mathit{lappend}'\ xs'\ ys))).
\end{array}
$$

By (9) we must show for arbitrary resolution i and functions g and h, that

$$
\begin{array}{l}
(\forall\, xs\ ys\,.\,\mathit{ltake}\,i\,(g\,xs\,ys) = \mathit{ltake}\,i\,(h\,xs\,ys)) \longrightarrow \\
\quad (\forall\, xs\ ys\,.\,\mathit{ltake}\,(i+1)\,(F\,g\,xs\,ys) = \mathit{ltake}\,(i+1)\,(F\,h\,xs\,ys)).
\end{array}
$$

So we take arbitrary i, xs, and ys, and prove

$$
\mathit{ltake}\,(i+1)\,(F\,g\,xs\,ys) = \mathit{ltake}\,(i+1)\,(F\,h\,xs\,ys)
$$

assuming we have $(\forall\, xs\ ys\,.\,\mathit{ltake}\,i\,(g\,xs\,ys) = \mathit{ltake}\,i\,(h\,xs\,ys))$. There are two cases to consider, depending on whether xs is empty or not:

case $xs = []$:

$$
\begin{array}{ll}
 & \mathit{ltake}\,(i+1)\,(F\,g\,xs\,ys) = \mathit{ltake}\,(i+1)\,(F\,h\,xs\,ys) \\
\Leftrightarrow & \mathit{ltake}\,(i+1)\,(F\,g\,[]\,ys) = \mathit{ltake}\,(i+1)\,(F\,h\,[]\,ys) \\
\Leftrightarrow & \mathit{ltake}\,(i+1)\,(\text{case}\,[]\,\text{of} \\
 & \qquad\qquad [] \Rightarrow ys \\
 & \qquad\quad \mid (x\,\#\,xs') \Rightarrow x\,\#\,(g\,xs'\,ys)) = \\
 & \mathit{ltake}\,(i+1)\,(\text{case}\,[]\,\text{of} \\
 & \qquad\qquad [] \Rightarrow ys \\
 & \qquad\quad \mid (x\,\#\,xs') \Rightarrow x\,\#\,(h\,xs'\,ys))
\end{array}
$$

$\Leftrightarrow$ $ltake\,(i+1)\,ys = ltake\,(i+1)\,ys$
$\Leftrightarrow$ True.

case $xs = (x\#xs')$:

$ltake\,(i+1)\,(F\,g\,xs\,ys) = ltake\,(i+1)\,(F\,h\,xs\,ys)$
$\Leftrightarrow$ $ltake\,(i+1)\,(F\,g\,(x\#xs')\,ys) = ltake\,(i+1)\,(F\,h\,(x\#xs')\,ys)$
$\Leftrightarrow$ $ltake\,(i+1)\,(\text{case}\,(x\#xs')\,\text{of}$
$\quad [] \Rightarrow ys$
$\quad \mid (x\,\#\,xs') \Rightarrow x\,\#\,(g\,xs'\,ys)) =$
$ltake\,(i+1)\,(\text{case}\,(x\#xs')\,\text{of}$
$\quad [] \Rightarrow ys$
$\quad \mid (x\,\#\,xs') \Rightarrow x\,\#\,(h\,xs'\,ys))$
$\Leftrightarrow$ $ltake\,(i+1)\,(x\,\#\,(g\,xs'\,ys)) = ltake\,(i+1)\,(x\,\#\,(h\,xs'\,ys))$
$\Leftrightarrow$ $ltake\,i\,(g\,xs'\,ys) = ltake\,i\,(h\,xs'\,ys)$
$\Leftrightarrow$ True {by assumption}.

Thus we can conclude that *lappend* has a unique fixed point definition. We were able to carry out this proof in Isabelle in three steps, again taking about a second of CPU time.

5.2 Other CER combinators

Cer combinators can also be defined over product and sum types. The lazy list CER can be generalized to work over any coinductive type that has a notion of depth, such as coinductive trees. A more powerful function-space CER is discussed in Sect. 6.

5.3 Demonstrating equality between coinductive elements

Converging equivalence relations can also be useful in showing that two elements of a target space are equal. Axiom (6) (restated below) says that to show two target elements x and y are equal, one simply needs to show they are equivalent at all resolutions j

$$(\forall j\,.\,x \overset{j}{\approx} y) \longrightarrow x = y.$$

We can often demonstrate that x and y are equivalent at all resolutions by well-founded induction, since $(<)$ is a well-founded relation. For example, given two arbitrary lazy lists ys and zs, we can prove the following lemma about *lappend* by (simple) induction on i:

Lemma 4

$$\forall xs\,.\,ltake\,i\,(lappend\,(lappend\,xs\,ys)\,zs) = ltake\,i\,(lappend\,xs\,(lappend\,ys\,zs)).$$

Proof
case $i = 0$:
Take xs to be an arbitrary lazy list. Then
$ltake\,i\,(lappend\,(lappend\,xs\,ys)\,zs) = ltake\,i\,(lappend\,xs\,(lappend\,ys\,zs))$
$\Leftrightarrow$ $ltake\,0\,(lappend\,(lappend\,xs\,ys)\,zs) = ltake\,0\,(lappend\,xs\,(lappend\,ys\,zs))$
$\Leftrightarrow$ $[] = []$
$\Leftrightarrow$ True.

case $i = (k+1)$:

Induction hypothesis:

Assume $(\forall xs.\ \mathit{ltake}\ k\ (\mathit{lappend}\ (\mathit{lappend}\ xs\ ys)\ zs) = \mathit{ltake}\ k\ (\mathit{lappend}\ xs\ (\mathit{lappend}\ ys\ zs)))$

Take xs to be an arbitrary lazy list. Then

$\mathit{ltake}\ i\ (\mathit{lappend}\ (\mathit{lappend}\ xs\ ys)\ zs) = \mathit{ltake}\ i\ (\mathit{lappend}\ xs\ (\mathit{lappend}\ ys\ zs))$

$\Leftrightarrow\ (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (\mathit{lappend}\ xs\ ys)\ zs) = \mathit{ltake}\ (k+1)\ (\mathit{lappend}\ xs\ (\mathit{lappend}\ ys\ zs)))$

subcase $xs = []$:

$\Leftrightarrow\ (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (\mathit{lappend}\ []\ ys)\ zs) = \mathit{ltake}\ (k+1)\ (\mathit{lappend}\ []\ (\mathit{lappend}\ ys\ zs)))$

$\Leftrightarrow\ (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ ys\ zs) = \mathit{ltake}\ (k+1)\ (\mathit{lappend}\ ys\ zs))$

$\Leftrightarrow$ True.

subcase $xs = (x \# xs')$:

$\Leftrightarrow\ (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (\mathit{lappend}\ (x \# xs')\ ys)\ zs) = \mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (x \# xs')\ (\mathit{lappend}\ ys\ zs)))$

$\Leftrightarrow\ (\mathit{ltake}\ (k+1)\ (\mathit{lappend}\ (x \# (\mathit{lappend}\ xs'\ ys))\ zs) = \mathit{ltake}\ (k+1)\ (x \# (\mathit{lappend}\ xs'\ (\mathit{lappend}\ ys\ zs))))$

$\Leftrightarrow\ (\mathit{ltake}\ (k+1)\ (x \# (\mathit{lappend}\ (\mathit{lappend}\ xs'\ ys)\ zs)) = \mathit{ltake}\ (k+1)\ (x \# (\mathit{lappend}\ xs'\ (\mathit{lappend}\ ys\ zs))))$

$\Leftrightarrow\ (\mathit{ltake}\ k\ (\mathit{lappend}\ (\mathit{lappend}\ xs'\ ys)\ zs) = \mathit{ltake}\ k\ (\mathit{lappend}\ xs'\ (\mathit{lappend}\ ys\ zs)))$

$\Leftrightarrow$ True {by induction hypothesis}.

This proof took four steps in Isabelle, and relied on the following facts about *lappend*, each proved in two steps by expanding *lappend*'s recursive definition once and simplifying:

$$\mathit{lappend}\ []\ ys = ys$$
$$\mathit{lappend}\ (x\#xs)\ ys = x \# (\mathit{lappend}\ xs\ ys)$$

Given Lemma 4 and CER axiom (6) instantiated to the lazy list CER, we can then easily show in one Isabelle step that $\mathit{lappend}\ (\mathit{lappend}\ xs\ ys)\ zs = \mathit{lappend}\ xs\ (\mathit{lappend}\ ys\ zs)$.

6 Defining functions with unbounded look-ahead

The functions we have defined so far examine their arguments by performing at most one pattern match on a lazy list before producing an element of a result list. However, there is a class of functions that can examine a potentially infinite amount of their argument lists before deciding the next element to output. An example is the *lazy filter* function of type $(\alpha \to \mathit{bool}) \to \alpha\ \mathit{llist} \to \alpha\ \mathit{llist}$, which takes a predicate P and a lazy list xs, and returns a lazy list of the same type consisting only of those elements of xs satisfying P. A candidate set of recursion equations for this function might be

$$\begin{array}{lll} \mathit{lfilter}\ P\ [] & = [] & \\ \mathit{lfilter}\ P\ (x\#xs) & = \mathit{lfilter}\ P\ xs, & \text{if } \neg(P\ x) \\ \mathit{lfilter}\ P\ (x\#xs) & = x \# (\mathit{lfilter}\ P\ xs), & \text{if } P\ x \end{array}$$

Sadly, this intuitively appealing set of equations does not completely define *lfilter*. If *lfilter* is given an infinite list xs, none of whose elements satisfy P, then the above

equations do not specify what the result list should be. For example, the equations are satisfied if *lfilter* returns in this case the infinite list $[arb, arb, \ldots]$, where *arb* is an arbitrary element of the appropriate type. In other words, the equations do not have a unique solution.

Happily we can remedy the situation as follows: We define by induction over *nat* a predicate *firstPelemAt* of type $(\alpha \to bool) \to \alpha\ llist \to nat \to bool$. The expression $(firstPelemAt\ P\ xs\ i)$ is true if xs has at least $(i+1)$ elements and i is the position of the first element of xs satisfying P. We can then define the predicate *never* of type $(\alpha \to bool) \to \alpha\ llist \to bool$ as

$$never\ P\ xs \equiv \forall i\ .\ \neg(firstPelemAt\ P\ xs\ i)$$

which is true when there are no elements in xs satisfying P. If we modify the initial recursive equations as follows:

$$\begin{array}{lll} lfilter\ P\ xs & = [], & \text{if } never\ P\ xs \\ lfilter\ P\ (x\#xs) & = lfilter\ P\ xs, & \text{if } \neg(never\ P\ xs) \wedge \neg(P\ x) \\ lfilter\ P\ (x\#xs) & = x\ \#\ (lfilter\ P\ xs), & \text{if } \neg(never\ P\ xs) \wedge P\ x \end{array}$$

then the set of equations does indeed have a unique solution. This function is not computable, since the predicate *never* can scan an infinite number of elements, but it is nevertheless mathematically valid in HOL. The CERs described above are not powerful enough to prove this, but we can define a *well-founded function-space* CER combinator that is. Given a CER C with $(<)$ of type $\alpha \to \alpha \to bool$ and $(\approx)$ with type $\alpha \to \beta \to \beta \to bool$, and another well-founded transitive relation $(\prec)$ of type $\tau \to \tau \to bool$, we define our new CER C' with $(<')$ and $(\approx')$ as follows:

$$\begin{array}{l} (<') :: (\alpha * \tau) \to (\alpha * \tau) \to bool \\ (\approx') :: (\alpha * \tau) \to (\tau \to \beta) \to (\tau \to \beta) \to bool \end{array}$$

$$\begin{array}{ll} (a', t') <' (a, t) & \equiv a' < a \vee (a' = a \wedge t' \prec t) \\ g \overset{(a,t)}{\approx'} h & \equiv \forall a'\ t'\ .\ ((a', t') <' (a, t)) \vee ((a', t') = (a, t)) \longrightarrow (g\ t') \overset{a'}{\approx} (h\ t') \end{array}$$

It is a fair amount of work to show that C' is in fact a CER, and space constraints force us to elide the details.

Intuitively, however, C' allows us to generalize well-founded recursion in the following way: A well-founded recursive function is forced to have its argument decrease in size on every recursive call. With C', the function being defined is allowed a choice; it can either decrease the size of its argument when making a recursive call, or not decrease its argument size but then make sure the element it is returning is "better" than the element returned from its recursive call.

In the case of functions returning lazy lists, a "better" lazy list is one that looks just like the lazy list returned by the recursive call, but with at least one extra element added to the front.

For us to use C' on *lfilter*, we need to specify a suitable well-founded transitive relation $(\prec)$. The relation we choose is one that holds when the first element satisfying P occurs sooner on the left-hand argument than on the right-hand argument:

$$xs \prec ys \equiv firstPelem\ P\ xs < firstPelem\ P\ ys$$

where

$$\begin{array}{lll} firstPelem\ P\ xs & = 0, & \text{if } never\ P\ xs \\ & = 1 + (\varepsilon i\ .\ firstPelemAt\ P\ xs\ i), & \text{otherwise} \end{array}$$

We arbitrarily decide that a list containing no P-elements is $\prec$-smaller than any list with at least one P-element.

When analyzing the revised recursive equations for *lfilter*, if xs has no P-elements then we return immediately, otherwise xs has to have at least one P-element. If that element is not at the head of the list, then the tail of the list is $\prec$-smaller than xs. If the first P-element is at the head of xs, then the tail of the list is not $\prec$-smaller than xs, but the output list has one more element than the list returned by the recursive call. Thus we informally conclude that the *lfilter* is uniquely defined.

We have also proved this fact formally in Isabelle. After inductively proving various simple lemmas about *firstPelemAt*, *never*, and *firstPelem*, we were able to prove that *lfilter* is uniquely defined in five steps. We first translated the recursive equations above into a contracting function F. We used C' prove that F is contracting, first by expanding the definition of F and simplifying, and then by performing a case analysis (no induction required!) on whether the *nat* component of the current resolution was equal to zero. It took Isabelle two seconds to perform the proof.

Although we had to prove lemmas about *firstPelemAt*, *never*, and *firstPelem*, the proofs are not hard and it turns out we can reuse these results when defining other functions that perform unbounded search on lazy lists. For example, the *lflatten* function takes a lazy list of lazy lists, and flattens all of the elements into a single lazy list. The *lflatten* function can also be uniquely defined using *never*:

$$\begin{array}{lll} \mathit{lflatten}\ \mathit{xss} & = [], & \text{if } \mathit{never}\ (\lambda \mathit{xs}\ .\mathit{xs} \neq [])\ \mathit{xss} \\ \mathit{lflatten}\ (\mathit{xs}\#\mathit{xss}) & = \mathit{lappend}\ \mathit{xs}\ (\mathit{lflatten}\ \mathit{xss}), & \text{otherwise} \end{array}$$

The proof proceeds in Isabelle exactly as it does for *lfilter* except that we perform one additional case analysis on whether $xs = []$. The proof takes three seconds to complete.

7 Proof of the main result

Although the proof of the main theorem is too lengthy to describe here, we will provide a rough outline. Given a CER with resolution space α, target space β, well-founded relation $(<)$, indexed equivalence relation $(\approx)$, and an arbitrary contracting function F of type $\beta \to \beta$, the technique will be to construct an approximation map $\mathit{apx}\ F$ that converges globally to the desired fixed point. We then prove that this fixed point is unique by showing that any two fixed points of F are equal.

The function *apx* of type $(\beta \to \beta) \to \alpha \to \beta$ that builds an approximation map from a contracting function is defined by well-founded recursion on $(<)$ as follows:

$$\begin{array}{l} \mathit{apx}\ F\ i \equiv F\ (\mathit{local\_limit}\ (\mathit{cut}\ (\mathit{apx}\ F)\ i)\ i) \\ \text{where} \\ \qquad \mathit{cut}\ f\ i\ x \equiv \text{if } x < i \text{ then } f\ x \text{ else } \mathit{arbitrary}. \end{array}$$

At each resolution i, the function *apx* uses *local_limit* to obtain the best possible approximation of $\mathrm{fix}\ F$, given the approximations it has already computed at all lower resolutions. The result of calling *local_limit* may still not be close enough at resolution i, so *apx* maps the local limit through F, which will bring the result close enough. The helper function *cut* is used to ensure that the recursive call to $\mathit{apx}\ F$ is only made at lower resolutions than i, ensuring well-foundedness. If *local_limit* attempts to invoke $\mathit{cut}\ (\mathit{apx}\ F)\ i$ at any other resolution, then *cut* returns an arbitrary element instead.

Once we have proved by well-founded induction that *apx* is well defined, the next step is to establish that $\mathit{apx}\ F$ is convergent up to each resolution i. To do this we prove several lemmas, such as: if an approximation mapping f converges up to a local limit element z at resolution i, and also converges up to a local limit element z' at the same resolution, then z and z' are equivalent at all resolutions $i' < i$. With this, and the fact

that F is contracting, we can show that if $x \overset{i}{\approx} y$, then $F\,x \overset{i}{\approx} F\,y$. We then eventually show for all resolutions i that if $apx\,F$ converges up to local limit element $apx\,F\,i$ at resolution i, then $apx\,F\,i \overset{i}{\approx} F\,(apx\,F\,i)$. This lemma is the key to showing by well-founded induction over i that $apx\,F$ does in fact converge up to $apx\,F\,i$ at resolution i, and is also used to show that $global\_limit\,(apx\,F) \overset{i}{\approx} F\,(global\_limit\,(apx\,F))$ at each resolution i, and are thus equal by (6). This result establishes that a fixed point exists for F. We then show that any two fixed points x and y of F are equivalent at all resolutions by well-founded induction, and thus are equal, again by (6).

8 Conclusion

Related work The support for and application of well-founded induction and general coinduction has seen wide acceptance in the HOL theorem proving community. The well-founded definition package TFL used in HOL98 and Isabelle was written by Slind[15]. It can handle nested pattern matching in rule definitions, nested recursion in function bodies, and generates custom induction rules for each definition[16]. The PVS theorem prover[14] also uses well-founded induction as a basic definitional principle. A general theory of inductive and coinductive sets in Isabelle was developed by Paulson[12], based on least and greatest fixed points of monotone set-transforming functions, as well as a package for defining new inductive and coinductive sets by user-given introduction rules. The package avoids syntactic restrictions in the introduction rules by reasoning about each rule's underlying set-transformer semantics.

Paulson's Isabelle theories were applied by Frost[3] to formalize the static and dynamic semantics of a small functional language and prove that the two semantics were consistent with each other. Recursive functions are represented by infinitely nested environments, requiring consistency to be proved by coinduction. The language and proof, as well as the concept of coinduction as a variant of fixpoint induction, were introduced by Milner and Tofte[8].

A coinductive theory of streams (infinite-only lists) was developed by Miner[9] in the PVS theorem prover. Miner used this theory to model synchronous hardware circuits as corecursively-defined stream transformers. Using coinduction, he was able to optimize the implementation of a fault-tolerant clock synchronization circuit and a floating-point division circuit. In several cases a subcircuit was replaced by an optimized subcircuit, and the correctness of the replacement depended on non-trivial environmental assumptions in the surrounding circuit. Coinduction was used to verify the environmental assumptions and to show that the subcircuits were equivalent under the assumed environment.

A well-known alternative to coinductive types is the mathematical framework of *pointed complete partial orders* and *continuous functions*, also known as *domain theory*[5, 17]. This theory is supported by the HOLCF[10] object-logic in Isabelle, and also allows one to define infinite data structures such as lazy lists and trees. A wide variety of functions over these structures can then be recursively defined. The primary disadvantage of this approach is that one must add "extra" bottom-elements to the structures being defined. These extra elements are used to indicate that a function is non-terminating on its arguments. For example, the lazy filter function *lfilter* can be defined recursively in HOLCF, but the expression $lfilter\,P\,xs$ returns $\bot$ instead of [] when xs is an infinite list containing no elements satisfying P. Also, only so-called *admissible* predicates can be reasoned about inductively in domain theory, and it can be quite challenging to prove that a desired predicate is admissible. A comparison of the HOLCF approach to several other encodings of lazy lists is presented by Devillers et al[2].

Metric spaces[13] and topology[1] are another well-established definition mechanism. The notions of Cauchy sequences, complete metric spaces, and contractions inspired much of this work. We have not worked out the exact relationship between converging equivalence relations and Cauchy metric spaces; although one can construct a distance function for every CER based on the *nat* resolution space, it is not clear that distance functions can be always be constructed for more complex resolution spaces. Also, the conditions under which a function F is contracting in a CER seem to be less restrictive than the corresponding conditions in a metric space. More importantly from a verification perspective, well-founded induction seems easier to apply in current theorem provers than does the continuous mathematics required for metric spaces.

Current and future work We are currently using CERs to specify and reason about processor microarchitectures as recursively defined stream transformers. This work is part of the Hawk project[7], which is developing a domain-specific functional language for specifying, simulating, and reasoning about such microarchitectures at a high level of abstraction. We have been able to use CERs and the unique fixed point lemmas in Sect. 3.2 to develop a domain-specific *microarchitecture algebra*[6] in Isabelle, which we use to verify Hawk specifications.

We have mechanized the theory of CERs in Isabelle and have been able to define interesting lazy functions recursively, such as zip, filter, flatten, and several microarchitecture specifications. However, we did so by reasoning about unique fixed points directly. One possibility would be to write a package along the lines of TFL where users need only supply a system of pattern matching recursive equations and a CER. The package would then automate the unique existence proofs.

We have not yet seriously explored nested recursion with CERs, but we would like to in the future.

Although we have defined CERs over streams and lazy lists, many structures in language semantics and process algebras can be seen as coinductive trees. It would be interesting to define some of these structures recursively and reason about them inductively, as we did for *lappend* in Sect. 5.3.

9 Acknowledgements

We wish to thank Byron Cook, Sava Krstić, and John Launchbury for their valuable contributions to this research. The author is supported by a graduate research fellowship with the National Science Foundation, and grants from the Air Force Material Command (F19628-93-C-0069) and Intel Strategic CAD Labs.

References

1. Buskes, G., and van Rooij, A. *Topological Spaces: from distance to neighborhood.* UTM Series. Springer, New York, 1997.
2. Devillers, M., Griffioen, D., and Müller, O. Possibly infinite sequences in theorem provers: A comparative study. In *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97* (Murray Hill, NJ, Aug. 1997), vol. 1275 of *LNCS*, Springer-Verlag, pp. 89–104.
3. Frost, J. A case study of co-induction in Isabelle. Tech. Rep. 359, University of Cambridge, Computer Laboratory, Feb. 1995. Revised version of CUCL 308, August 1993.
4. Gordon, M. J. C., and Melham, T. F. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

5. GUNTER, C. A. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing Science. The MIT Press, 1992.
6. MATTHEWS, J., AND LAUNCHBURY, J. Elementary microarchitecture algebra. To appear in CAV99, International Conference on Computer Aided Verification, July 1999.
7. MATTHEWS, J., LAUNCHBURY, J., AND COOK, B. Specifying microprocessors in Hawk. In *IEEE International Conference on Computer Languages* (Chicago, Illinois, May 1998), pp. 90–101.
8. MILNER, R., AND TOFTE, M. Co-induction in relational semantics. *Theoretical Computer Science 87* (1991), 209–220.
9. MINER, P. *Hardware Verification Using Coinductive Assertions.* PhD thesis, Indiana University, 1998.
10. MÜLLER, O., NIPKOW, T., V. OHEIMB, D., AND SLOTOSCH, O. HOLCF = HOL + LCF. To appear in Journal of Functional Programming, 1999.
11. PAULSON, L. *Isabelle: A Generic Theorem Prover.* Springer-Verlag, 1994.
12. PAULSON, L. C. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation 7*, 2 (Apr. 1997), 175–204.
13. RUDIN, W. *Principles of Mathematical Analysis*, 3 ed. McGraw-Hill, 1976.
14. RUSHBY, J., AND STRINGER-CALVERT, D. W. J. A less elementary tutorial for the PVS specification and verification system. Tech. Rep. SRI-CSL-95-10, SRI International, Menlo Park, CA, June 1995. Revised, July 1996.
15. SLIND, K. Function definition in higher order logic. In *Ninth international Conference on Theorem Proving in Higher Order Logics TPHOL* (Turku, Finland, Aug. 1996), J. Von Wright, J. Grundy, and J. Harrison, Eds., vol. 1125 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 381–398.
16. SLIND, K. Derivation and use of induction schemes in higher-order logic. *Lecture Notes in Computer Science 1275* (1997), 275–290.
17. TENNENT, R. D. *Semantics of Programming Languages.* Prentice Hall, New York, 1991.

DSL Implementation Using Staging and Monads

Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic
Pacific Software Research Center
Oregon Graduate Institute
P.O. Box 91000 Portland, Oregon 97291-1000 USA

March 21, 1999

Abstract

The impact of Domain Specific Languages (DSLs) on software design is considerable. They allow programs to be more concise than equivalent programs written in a high-level programming languages. They relieve programmers from making decisions about data-structure and algorithm design, and thus allows solutions to be constructed quickly. Because DSL's are at a higher level of abstraction they are easier to maintain and reason about than equivalent programs written in a high-level language, and perhaps most importantly *they can be written by domain experts rather than programmers.*

The problem is that DSL implementation is costly and prone to errors, and that high level approaches to DSL implementation often produce inefficient systems. By using two new programming language mechanisms, program staging and monadic abstraction, we can lower the cost of DSL implementations by allowing reuse at many levels. These mechanisms provide the expressive power that allows the construction of many compiler components as reusable libraries, provide a direct link between the semantics and the low-level implementation, and provide the structure necessary to reason about the implementation.

1 Introduction

We outline an improved *method* for the design and implementation of Domain-Specific Languages (DSLs). The method builds upon our experience with staged programming using the staged programming language MetaML [27, 26]. The method also incorporates ideas from other researchers in the areas of modular language design [28, 24, 12], correct compiler generation [15, 19, 18, 16, 10], and partial evaluation [8, 13]. While relying on recent advances in functional programming (such as higher-order type constructors, and local polymorphism), it is applicable to *all kinds of languages*, not just applicative ones. The method unifies many of these ideas into a coherent process.

A problem with the DSL approach to software construction is its cost. Realizing a DSL requires an implementation. Such implementations are large and expensive to produce. So, unless many solutions are required, it may not pay to build a compiler or other implementation mechanism. DSL implementation is also conceptually hard. Most software engineers are not comfortable taking on the task of language design and implementat5ion. Even if they are, language implementation is a difficult, complex process that does not easily scale. An implementation for a simple language often does not scale as the language evolves to meet newer demands. Lowering the cost of DSL implementations, and making good ones more manageable, will make the DSL approach applicable to a broader domain of problems.

Our approach to solving these problems is to apply new methods of abstraction such as monads [28, 31] and staging [27, 26] to the implementation of DSLs. This makes the effort required to

build a compiler for a DSL reusable and spreads the cost over several DSLs. To make language implementation manageable for the masses, there must exist good rules of thumb for language implementation. One way to accomplish this is by elaborating *a step by step method* that splits the labor into well-defined steps, each with a relatively small amount of work. In our method, each step deals with an orthogonal design decision. By using good abstraction principles, our method partitions each design decision into a separate code module. In addition, our method makes explicit the propositions that must be proved to show the correctness of the compiler with respect to its semantics.

Our method comprises the following steps. First, construct the denotational semantics as an interpreter in a functional language. Second, capture the effects of the language, and the environment in which the target language must run, in a monad. Then rewrite the interpreter in a monadic style. Third, stage the interpreter using meta-programming techniques. This staging is similar to the staging of interpreters using a partial evaluator, but is explicit rather than implicit, since the programmer places the annotations directly, rather than using an automatic binding time analysis to discover where they should be placed. This leaves programmers in complete control, and they can limit what appears in the residual program. Fourth, the resulting program is both a data-structure and a program, so it can be both directly executed and analyzed. This analysis can include both source to source transformations, or translation into another form (i.e. intermediate code or assembly language). Because the programmer has complete control over the earlier steps, the structure of the residual program is highly constrained, and this final translation can be a trivial task.

Staging of interpreters using partial evaluation has been done before [1, 5]. The contribution of this paper is to show that this can all be done in a single program. A system incorporating staging as a first class feature of a language is a powerful tool. While using such a tool to write a compiler the source language can be given semantics, it can be staged, translated, and optimized all in a single paradigm. It requires neither additional processes nor tools, and is under the complete control of the programmer; all the while maintaining a direct link between the semantics of interpreter and those of the compiler.

2 Staging in MetaML

METAML is almost a conservative extension of Standard ML. Its extensions include four staging annotations. To delay an expression until the next stage one places it between meta-brackets. Thus the expression `<23>` (pronounced "bracket 23") has type `<int>` (pronounced "code of int"). The annotation, ~*e* splices the deferred expression obtained by evaluating *e* into the body of a surrounding Bracketed expression; and `run` *e* evaluates *e* to obtain a deferred expression, and then evaluates this deferred expression. It is important to note that ~*e* is only legal within lexically enclosing Brackets. We illustrate the important features of the staging annotations in the short METAML sessions below.

```
-| val z = 3+4;
val z = 7 : int
```

Users access METAML through a *read-type-eval-print* top-level. The declaration for `z` is read, type-checked to see that it has a consistent type (`int` here), evaluated (to 7), and then both its value and type are printed.

```
-| val quad = ( 3+4,  <3+4>,    lift (3+4), <z>  );
```

```
val quad =    ( 7,    <3 %+ 4>, <7>,        <%z> ) :
              ( int * <int> *   <int> *     <int>)
```

The declaration for quad contrasts normal evaluation with the three ways objects of type code can be constructed. Placing brackets around an expression (<3+4>) defers the computation of 3+4 to the next stage, returning a piece of code. Lifting an expression (lift (3+4)) evaluates that expression (to 7 here) and then lifts the value to a piece of code that when evaluated returns the same value. Brackets around a free variable (<z>) creates a new constant piece of code with the value of the variable. Such constants print with a % sign to indicate they are constants. We call this *lexical-capture* of free variables. Because in METAML operators (such as + and *) are also identifiers, free occurrences of operators in constructed code often appear with % in front of them.

```
-| fun inc x = <1 + ~x>;
val inc = Fn  : ['a].<int> -> <int>
```

The declaration of the function inc illustrates that larger pieces of code can be constructed from smaller ones by using the escape annotation. Bracketed expressions can be viewed as *frozen*, i.e. evaluation does not apply under brackets. However, is it often convenient to allow some reduction steps inside a large frozen expression while it is being constructed, by "splicing" in a previously constructed piece of code. METAML allows one to *escape* from a frozen expression by prefixing a sub-expression within it with the tilde (~) character. Escape must only appear inside brackets.

```
-| val six = inc <5>;
val six =  <1 %+ 5> : <int>
```

In the declaration for six, the function increment is applied to the piece of code <5> constructing the new piece of code <1 %+ 5>.

```
-| run six;
val it = 6 : int
```

Running a piece of code, strips away the enclosing brackets, and evaluates the expression inside. To give a brief feel for how MetaML is used to construct larger pieces of code at run-time consider:

```
-| fun mult x n = if n=0 then <1> else < ~x * ~(mult x (n-1)) >;
val mult = fn  : <int> -> int  -> <int>

-| val cube = <fn y => ~(mult <y> 3)>;
val cube = <fn a => a * (a * (a * 1))> : <int  -> int>

-| fun exponent n = <fn y => ~(mult <y> n)>;
val exponent = fn  : int  -> <int  -> int>
```

The function mult, given an integer piece of code x and an integer n, produces a piece of code that is an n-way product of x. This can be used to construct the code of a function that performs the cube operation, or generalized to a generator for producing an exponentiation function from a given exponent n. Note how the looping overhead has been removed from the generated code. This is the purpose of program staging and it can be highly effective as discussed elsewhere [4, 6, 11, 23, 27].

3 Monads in METAML

We assume the reader has a working knowledge of monads[29, 31]. We use the *unit* and *bind* formulation of monads[30]. In METAML a monad is a data structure encapsulating a type constructor M and the *unit* and *bind* functions.

```
datatype ('M : * -> * ) Monad = Mon of
   (['a]. 'a -> 'a 'M) *                          (* unit function *)
   (['a,'b]. 'a 'M -> ('a -> 'b 'M) -> 'b M);  (* bind function *)
```

This definition uses SML's postfix notation for type application, and two non-standard extensions to ML. First, it declares that the argument `('M : * -> * )` of the type constructor `Monad` is itself a unary type constructor [7]. We say that `'M` has *kind*: `* -> *`. Second, it declares that the arguments to the constructor `Mon` must be polymorphic functions [17]. The type variables in brackets, e.g. `['a,'b]`, are universally quantified. Because of the explicit type annotations in the `datatype` definitions the effect of these extensions on the Hindley-Milner type inference system is well known and poses no problems for the METAML type inference engine.

In METAML, `Monad` is a first-class, although *pre-defined* or *built-in* type. In particular, there are two syntactic forms which are aware of the `Monad` datatype: `Do` and `Return`. `Do` and `Return` are METAML's syntactic interface to the *unit* and *bind* of a monad. We have modeled them after the do-notation of Haskell[9, 20]. An important difference is that METAML's `Do` and `Return` are both parameterized by an expression of type `'M Monad`. `Do` and `Return` are syntactic sugar for the following:

```
(*      Syntactic Sugar                    Derived Form     *)

Do (Mon(unit,bind)) { x <- e; f }   =   bind e (fn x => f)

Return (Mon(unit,bind)) e           =   unit e
```

In addition the syntactic sugar of the `Do` allows a sequence of $\texttt{x}_i$ `<-` $\texttt{e}_i$ forms, and defines this as a nested sequence of `Do`'s. For example:

```
Do m { x1 <- e1; x2 <- e2 ; x3 <- e3 ; e4 }   =
   Do m { x1 <- e1; Do m { x2 <- e2 ; Do m { x3 <- e3 ; e4 }}}
```

Users may freely construct their own monads, though they should be very careful that their instantiation meets the monad axioms. The monad axioms, expressed in METAML's `Do` and `Return` notation are:

```
Do { x <- Return e ; z }          = z[e/x]
Do { x <- m ; Return x }          = m
Do { x <- Do { y <- a ; b } ; c } = Do { y' <- a ; Do { x <- b[y'/y] ; c } }
                                  = Do { y' <- a ; x <- b[y'/y] ; c }
```

4 Illustrating our compiler development method

In this section, we illustrate our method by building the front end of a compiler for a small imperative *while-language*. We proceed in three steps. First, we introduce the language and its denotational semantics by giving a monadic interpreter as a one stage METAML program. Second, we stage this interpreter by using a two stage METAML program in order to produce a compiler. Third, we illustrate the usefulness of the staging approach, by showing how using MetaML's intensional analysis tools can be used to optimize or further translate the output of a staged program.

4.1 The while-language

In this section, we introduce a simple *while-language* composed from the syntactic elements: expressions (`Exp`) and commands (`Com`). In this simple language expressions are composed of integer constants, variables, and operators. A simple algebraic datatype to describe the abstract syntax of expressions is given in METAML below:

```
datatype Exp =
  Constant of int                (*   5      *)
| Variable of string             (*   x      *)
| Minus of (Exp * Exp)           (*  x - 5   *)
| Greater of (Exp * Exp)         (*  x > 1   *)
| Times of (Exp * Exp) ;         (*  x * 4   *)
```

Commands include assignment, sequencing of commands, a conditional (*if* command), while loops, a print command, and a declaration which introduces new statically scoped variables. A declaration introduces a variable, provides an expression that defines its initial value, and limits its scope to the enclosing command. A simple algebraic datatype to describe the abstract syntax of commands is:

```
datatype Com =
  Assign of (string * Exp)          (*  x := 1                          *)
| Seq of (Com * Com)                (*  { x := 1; y := 2 }              *)
| Cond of (Exp * Com * Com)         (*  if x then x := 1 else y := 1    *)
| While of (Exp * Com)              (*  while x>0 do x := x - 1         *)
| Declare of (string * Exp * Com)   (*  declare x = 1 in x := x - 1     *)
| Print of Exp;                     (*  print x                          *)
```

A simple while-program in concrete syntax, such as

```
declare x = 150 in
   declare y = 200 in { while x > 0 do { x := x - 1; y := y - 1 }; print y}
```

is encoded abstractly in these datatypes as follows:

```
val S1 =
Declare("x",Constant 150,
  Declare("y",Constant 200,
    Seq(While(Greater(Variable "x",Constant 0),
              Seq(Assign("x",Minus(Variable "x",Constant 1)),
                  Assign("y",Minus(Variable "y",Constant 1)))),
        Print(Variable "y"))));
```

4.2 The structure of the solution

Staging is an important technique for developing efficient programs, but it requires some forethought. To get the best results one should design algorithms with their staged solutions in mind.

The meaning of a while-program depends only on the meaning of its component expressions and commands. In the case of expressions, this meaning is a function from environments to integers. The environment is a mapping between names (which are introduced by `Declare`) and their values.

There are several ways that this mapping might be implemented. Since we intend to stage the interpreter, we break this mapping into two components. The first component, a list of names, will be completely known at compile-time. The second component, a list of integer values that behaves like a stack, will only be known at the run-time of the compiled program.

The functions that access this environment distribute their computation into two stages. First, determining at what location a name appears in the name list, and second, by accessing the correct integer from the stack at this location. In a more complicated compiler the mapping from names to locations would depend on more than just the declaration nesting depth, but the principle remains the same. Since every variable's location can be completely computed at compile-time, it is important that we do so, and that these locations appear as constants in the next stage.

Splitting the environment into two components is a standard technique (often called a binding time improvement) used by the partial evaluation community[8]. We capture this precisely by the following purely functional implementation.

```
type location = int;
type index = string list;
type stack = int list;

(* position : string -> index -> location *)
fun position name index =
    let fun pos n (nm::nms) = if name = nm then n else pos (n+1) nms
    in pos 1 index end;

(* fetch : location -> stack -> int *)
fun fetch n (v::vs) = if n = 1 then v else fetch (n-1) vs;

(* put: location -> int -> stack -> stack *)
fun put n x (v::vs) = if n = 1 then x::vs else v::(put (n-1) x vs);
```

The meaning of Com is a stack transformer and an output accumulator. It transforms one stack (with values of variables in scope) into another stack (with presumably different values for the same variables) while accumulating the output printed by the program.

To produce a monadic interpreter we could define a monad which encapsulates the index, the stack, and the output accumulation. Because we intend to stage the interpreter we do not encapsulate the index in the monad. We want the monad to encapsulate only the dynamic part of the environment (the stack of values where each value is accessed by its position in the stack, and the output accumulation).

The monad we use is a combination of *monad of state* and the *monad of output.*

```
datatype 'a M = StOut of (stack -> ('a * stack * string));
fun unStOut (StOut f) = f;
fun unit x = StOut(fn n => (x,n,""));
fun bind e f = StOut(fn n => let val (a,n1,s1) = (unStOut e) n
                                 val (b,n2,s2) =  unStOut(f a) n1
                             in (b,n2,s1 ^ s2) end);
val mswo : M Monad = Mon(unit,bind); (* Monad of state with output *)
```

The non-standard morphisms must describe how the stack is extended (or shrunk) when new variables come into (or out of) scope; how the value of a particular variable is read or updated; and how the printed text is accumulated. Each can be thought of as an action on the stack of mutable variables, or an action on the print stream.

```
(* read : location -> int M *)
fun read i = StOut(fn ns => (fetch i ns,ns,""));

(* write : location -> int -> unit  M *)
```

```
fun write i v = StOut(fn ns =>( (), put i v ns, "" ));

(* push: int -> unit  M *)
fun push x = StOut(fn ns => ( (), x :: ns, ""));

(* pop : unit M *)
val pop = StOut(fn (n::ns) => ((), ns, ""));

(* output: int -> unit M *)
fun output n = StOut(fn ns => ( (), ns, (toString n)^" "));
```

4.3 Step 1: monadic interpreter

Because expressions do not alter the stack, or produce any output, we could give an evaluation function for expressions which is not monadic, or which uses a simpler monad than the monad defined above. We choose to use the monad of state with output throughout our implementation for two reasons. One, for simplicity of presentation, and two because if the while language semantics should evolve, using the same monad everywhere makes it easy to reuse the monadic evaluation function with few changes.

The only non-standard morphism evident in the **eval1** function is **read**, which describes how the value of a variable is obtained. The monadic interpreter for expressions takes an index mapping names to locations and returns a computation producing an integer.

```
(*  eval1: Exp -> index -> int M *)
fun eval1 exp index =
case exp of
  Constant n => Return mswo n
| Variable x => let val loc = position x index
                in read loc end
| Minus(x,y) =>
   Do mswo { a <- eval1 x index ;
             b <- eval1 y index;
             Return mswo (a - b) }
| Greater(x,y) =>
   Do mswo { a <- eval1 x index ;
             b <- eval1 y index;
             Return mswo (if a '>' b then 1 else 0) }
| Times(x,y) =>
   Do mswo { a <- eval1 x index ;
             b <- eval1 y index;
             Return mswo (a * b) };
```

The interpreter for **Com** uses the non-standard morphisms **write**, **push**, and **pop** to transform the stack and the morphism **output** to add to the output stream.

```
(* interpret1 : Com -> index -> unit M *)
fun interpret1 stmt index =
case stmt of
  Assign(name,e) =>
   let val loc = position name index
   in Do mswo { v <- eval1 e index ;  write loc v } end
| Seq(s1,s2) =>
   Do mswo { x <- interpret1 s1 index;
```

```
            y <- interpret1 s2 index;
            Return mswo () }
| Cond(e,s1,s2) =>
   Do mswo { x <- eval1 e index;
            if x=1
               then interpret1 s1 index
               else interpret1 s2 index }
| While(e,body) =>
   let fun loop () =
        Do mswo { v <- eval1 e index ;
                  if v=0 then Return mswo ()
                         else Do mswo { interpret1 body index ;
                                        loop () } }
   in loop () end
| Declare(nm,e,stmt) =>
   Do mswo { v <- eval1 e index ;
            push v ;
            interpret1 stmt (nm::index);
            pop }
| Print e =>
   Do mswo { v <- eval1 e index;
            output v };
```

Although `interpret1` is fairly standard, we feel that two things are worth pointing out. First, the clause for the `Declare` constructor, which calls `push` and `pop`, implicitly changes the size of the stack and explicitly changes the size of the index (`nm:index`), keeping the two in synch. It evaluates the initial value for a new variable, extends the index with the variables name, and the stack with its value, and then executes the body of the `Declare`. Afterwards it removes the binding from the stack (using `pop`), all the while implicitly threading the accumulated output. The mapping is in scope only for the body of the declaration.

Second, the clause for the `While` constructor introduces a local tail recursive function `loop`. This function emulates the body of the while. It is tempting to control the recursion introduced by the `While` by using the recursion of the `interpret1` function itself by using a clause something like:

```
| While(e,body) =>
   Do mswo { v <- eval1 e index ;
            if v=0 then Return mswo ()
                   else Do mswo { interpret1 body index ;
                                  interpret1 (While(e,body)) index }
         }
```

Here, if the test of the loop is true, we run the body once (to transform the stack and accumulate output) and then repeat the whole loop again. This strategy, while correct, will have disastrous results when we stage the interpreter, as it will cause the first stage to loop infinitely.

There are two recursions going on here. First the unfolding of the finite data structure which encodes the program being compiled, and second, the recursion in the program being compiled. In an unstaged interpreter a single loop suffices. In a staged interpreter, both loops are necessary. In the first stage we only unfold the program being compiled and this must always terminate. Thus we must plan ahead as we follow our three step process. Nevertheless, despite the concessions we have made to staging, this interpreter is still clear, concise and describes the semantics of the while-language in a straight-forward manner.

4.4 Step 2: staged interpreter

To specialize the monadic interpreter to a given program we add two levels of staging annotations. The result of the first stage is the intermediate code, that if executed returns the value of the program. The use of the bracket annotation enables us to describe precisely the code that must be generated to run in the next stage. Escape annotations allow us to escape the recursive calls of the interpreter that are made when compiling a while-program.

```
(* eval2: Exp -> index -> <int M> *)
fun eval2 exp index =
case exp of
  Constant n => <Return mswo ~(lift n)>
| Variable x =>
   let val loc = position x index
   in <read ~(lift loc)> end
| Minus(x,y) =>
   <Do mswo { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswo (a - b) }>
| Greater(x,y) =>
   <Do mswo { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswo (if a '>' b then 1 else 0) }>
| Times(x,y) =>
   <Do mswo { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswo (a * b) }>;
```

The `lift` operator inserts the value of `loc` as the argument to the **read** action. The value of `loc` is known in the first-stage (compile-time), so it is transformed into a constant in the second-stage (run-time) by `lift`.

To understand why the escape operators are necessary, let us consider a simple example: `eval2 (Minus(Constant 3,Constant 1)) []`. We will unfold this example by hand below:

```
eval2 (Minus(Constant 3,Constant 1)) [] =

< Do mswo
     { a <- ~(eval2 (Constant 3) []);
       b <- ~(eval2 (Constant 1) []);
       Return mswo (a-b)} >                    =

< Do mswo
     { a <- ~<Return mswo 3>;
       b <- ~<Return mswo 1>;
       Return mswo (a - b)} >                  =

< Do mswo
     { a <- Return mswo 3;
       b <- Return mswo 1;
       Return mswo (a - b)} >                  =

< Do %mswo
     { a <- Return %mswo 3;
```

```
    b <- Return %mswo 1;
    Return %mswo (a %- b)} >
```

Each recursive call produces a bracketed piece of code which is spliced into the larger piece being constructed. Recall that escapes may only appear at level-1 and higher. Splicing is axiomatized by the reduction rule: ~<x> $\longrightarrow$ x, which applies only at level-1. The final step, where mswo and - become %mswo and %-, occurs because both are free variables and are lexically captured.

Interpreter for Commands.

Staging the interpreter for commands proceeds in a similar manner:

```
(*  interpret2 : Com -> index ->  <unit M> *)
fun interpret2 stmt index =
case stmt of
  Assign(name,e) =>
   let val loc = position name index
   in <Do mswo { n <- ~(eval2 e index) ;
                 write ~(lift loc) n }>
   end
| Seq(s1,s2) =>
   <Do mswo { x <- ~(interpret2 s1 index);
              y <- ~(interpret2 s2 index);
              Return mswo () }>
| Cond(e,s1,s2) =>
   <Do mswo { x <- ~(eval2 e index);
              if x=1
                 then ~(interpret2 s1 index)
                 else ~(interpret2 s2 index)}>
| While(e,body) =>
   <let fun loop () =
            Do mswo { v <- ~(eval2 e index);
                      if v=0
                         then Return mswo ()
                         else Do mswo { q <- ~(interpret2 body index); loop ()}
                    }
    in loop () end>
| Declare(nm,e,stmt) =>
   <Do mswo { x <- ~(eval2 e index) ;
              push x ;
              ~(interpret2 stmt (nm::index)) ;
              pop }>
| Print e =>
   <Do mswo { x <- ~(eval2 e index) ;
              output x }>;
```

4.4.1 An example.

The function interpret2 generates a piece of code from a Com object. To illustrate this we apply it to the simple program: declare x = 10 in { x := x - 1; print x } and obtain:

```
<Do %mswo
    { a <- Return %mswo 10
```

```
; %push a
; Do %mswo
    { e <- Do %mswo
              { d <- Do %mswo
                        { b <- %read 1
                        ; c <- Return %mswo 1
                        ; Return %mswo b %- c
                        }
              ; %write 1 d
              }
    ; g <- Do %mswo
              { f <- %read 1
              ; %output f
              }
    ; Return %mswo ()
    }
; %pop
}>
```

Note that the staged program is essentially a compiler, translating the syntactic representation of the while-program into the above monadic object-program that will compute its meaning. Note that in the object-program all of the compile-time operations have disappeared. This object-program is fully executable. Simply by using the run operator of METAML, it can be executed for prototyping purposes.

5 Step 3: Back-end translation and intermediate code optimization

METAML is a meta-programming system. It has an object language and a meta-language. Meta-programs are programs that manipulate object programs. In METAML both the object language and the meta-language are ML. In METAML an object-program is both a data structure that can be manipulated, and a program that can be run.

This duality plays an important role in target code generation. The result of applying the staged interpreter from the previous step (a meta-program) to a DSL program to be compiled is a highly constrained residual program (an object program). This program is both a data-structure and a program, so it can be both directly executed (rapid prototype) and analyzed.

We use the object-code analysis capabilities of MetaML to transform the object program into the final target language. This analysis can include both source to source transformations, or translation into another form (i.e. intermediate code, assembly language, or target language).

Control over the form of the residual program is crucial here. The residual program is always an ML program (ML is the object language). But the user can control the form of this ML program. A goal of the translation is to make the object program use only those ML features directly supported by the target language. For example, we may structure the staged interpreter such that the residual program is first order, or just a sequence of primitive actions encoded as non-standard morphisms in the monad. This is where we connect the abstract monadic actions to their efficient implementations.

The object program produced above is an ML code fragment. It can be executed or analyzed. The code produced by `interpret2` is a restricted subset of ML. Disregarding the higher-order functions implicit in the monad, it is first order, and contains only `Do` expressions, `Return` expres-

sions, `if` expressions, calls to the non-standard morphisms `read`, `write`, `push` , `pop`, and `output`, primitive arithmetic operators `-` and `'>'`, and local looping functions (like `loop` above). The code is so regular that it can be captured by a simple grammar. The next step is to analyze this code to make the final translation to the target language, or to apply some ML-source to ML-source level optimizations. The reader might notice that the object-program above could be considerably, further simplified by applying the monad laws. There are many opportunities for doing so. After these laws are applied we obtain the much more satisfying:

```
<Do %mswo
    { %push 10
    ; a <- %read 1
    ; b <- Return %mswo a %- 1
    ; c <- %write 1 b
    ; d <- %read 1
    ; e <- %output d
    ; Return %mswo ()
    ; %pop
    }>
```

In addition to the monad laws which hold for all monads, we can also use laws which hold for particular non-standard morphisms. For instance, in the example above, we could avoid the second read of location 1 using the following rule:

```
Do { e1; c <- %write 1 b ; d <- %read 1; e2} = Do { e; c <- %write 1 b; e2[b/d]}
```

Every target language will have many such laws, and because our target language is both executable-code, and data-structure we can perform these optimizations. The final step is to translate the ML code fragment into the target language. This step uses the same intensional analysis of code capabilities of the optimization steps, and is the subject of the next section.

5.1 Intensional analysis of code fragments

In this section, we outline how we do intensional analysis of residual code. We provide a high-level pattern matching based interface. Code patterns can be constructed by placing brackets around code. For example a pattern that matches the literal `5` can be constructed by:

```
-| fun is5 <5> = true
     | is5 _ = false;
val is5 = fn  : <int> -> bool

-| is5 (lift (1+4));
val it = true  : bool

-| is5 <0>;
val it = false  : bool
```

The function `is5` matches its argument to the constant pattern `<5>` if it succeeds it returns `true` else `false`. Pattern variables in code patterns are indicated by escaping variables in the code pattern.

```
-| fun parts < ~x + ~y > = SOME(x,y)
     | parts _ = NONE;
```

```
val parts = fn  : <int> -> (<int> * <int>) option

-| parts <6 + 7>;
val it = SOME (<6>,<7>) : (<int> * <int>) option

-| parts <2>;
val it = NONE  : (<int> * <int>) option
```

The function `parts` matches its argument against the pattern `< ~x + ~y >`. If its argument is a piece of code which is the sum of two sub terms, it binds the pattern variable `x` to the left subterm and the pattern variable `y` to the right subterm.

We use of higher-order pattern variables[22, 21] for code patterns that contain binding occurrences, such as lambda expressions, let expressions, do expressions, or functions.

For example, a high-order pattern that matches the code of a function `<fn x => ...>`, of type `<'a -> 'b>` is written in eta-expanded form `<fn x => ~(g <x>)>`. When the pattern matches, the matching binds the higher-order pattern variable `g` to a function with type `<'a> -> <'b>`

Every higher order pattern variable must be in fully saturated form, by applying it to all the bound variables of the code pattern. For example if `g` is a higher-order pattern variable with type `<'a> -> <'b> -> <'c>` then we must write `(~g <x> <y>)`. The arguments to the higher-order pattern variable must be explicit bracketed variables, one for each variable bound in the code pattern at the context where the higher-order pattern appears. A higher-order pattern variable is used like a function on the right-hand side of a matching construct.

For example functions which implement the three monad axioms are written as follows:

```
fun monad1 <do mswo { x <- return mswo ~e; ~(z <x>) }> = z e

fun monad2 <do mswo { x <- ~m; return x }> = m

fun monad3 <do mswo { x <- do mswo {y <- ~a; ~(b <y>)}; ~(c <x> }> =
   <do mswo { y' <- ~a; do mswo { z <- ~(b <y'>); ~(c <z>) }}>
```

When the the function `monad1` is applied to the code `<do mswo {a <- return mswo (g 3); h(a + 2)}>`, the pattern variable `e` is bound to the function `fn x => <h(~x + 2)>` which has the type `<int> -> <int M>`. The right-hand side of `monad1` rebuilds a new code fragment, substituting formal parameter `x` of `e` by `<g 3>`, constructing the code `<h((g 3)+ 2)>`.

This technique can be used to build optimizations, or to translate a residual program into a target language.

6 Conclusion

The important issues of efficient language implementation by refinement from high-level specifications are: the efficient use of the underlying target environment, and removing the layer of interpretative computation introduced by such specifications. We have shown that monads and staging are the right abstraction mechanisms to accomplish the task. To effectively use these tools we propose that DSL implementers follow a well defined method. We reiterate our method here:

- **Domain analysis.** The problem domain is analyzed to find the common abstractions around which the language is designed. This step is perhaps the most important step in a good language design. It has been studied extensively by others [32, 2, 3]. Our research group has been investigating the integration of DSL design and domain analysis for several years.

Recently Widen and Hook have summarized a "top level" view of this integration, which is called the Software Design Automation (SDA) method [33]. This method provides a design process and many synthesis techniques to facilitate the integration of traditional domain analysis activities with language design and implementation. The method we propose can be used in the context of SDA. It specifically addresses the language implementation phase of the process.

- **Definitional interpreter.** Once the language has been identified, the next step is to provide it with a semantics given as a pure functional interpreter. This program can be thought of as its high-level definition [14, 25]. high-level interpreters are usually easy to construct and provide a reference which can be consulted to resolve any ambiguity in the language specification discovered in further steps. By building it in an executable framework (a functional language, such as Haskell or ML) it also provides a rapid prototype against which expectations can be measured.

- **Binding time improvements.** The next step requires a binding separation [8]. By identifying compile-time versus run-time data structures in the definitional interpreter, we can separate those with *both* components into separate data-structures. Examples of binding time improvements include the separation of environments, which map names to values, into a compile-time index and a run-time stack, and the introduction of a local recursive function to separate the recursion which drives the analysis of the syntax of the program being interpreted from the recursion that encodes the looping of the `while` command.

- **Target domain analysis.** The next step is to analyze the target language to identify the primitive implementation features that will support the translation. This step is usually straight-forward as the target language is often fixed, and well understood.

- **Design a monad.** The next step is to design a monad to capture the effects and actions implicit in the target language. This is a hard step in the process since it requires both abstract knowledge about the structure and properties of monads, and detailed concrete knowledge about the target domain. The choices made in this step influence the structure of the monad, the structure of the monadic interpreter, and the run-time system which interacts with the low-level effects of the target language.

  Once the monad is designed, an implementation for the monad as a pure functional emulation must be produced. The implementation must emulate the actions in a purely functional setting by explicitly threading abstract representations of the actions such as "stores", "I/O streams", or "exception continuations" in and out of all computations.

- **Monadic Interpreter.** The next step is to refine the purely functional definitional interpreter into one written in a monadic style [28, 24, 13]. This implementation is still purely functional because the actions of the monad are emulated in a functional style. But because the actions are now explicit, we have moved the form of definition closer to the target language. This step often requires a big change to the structure of the source code, because the monad makes implicit much of the "plumbing" explicit in the interpreter. The cost of this restructuring is not without benefit. The removal of the explicit plumbing results in programs which are simpler, and more immune to future changes.

- **Staging.** The next step completes the binding-time separation begun in the binding time improvement step. That step separated the compile-time *data* from the run-time data. Staging separates the compile-time *computations* from the run-time computations. This is done

by placing explicit staging annotations in the program written in METAML. Staging is the crucial step that differentiates an (inefficient) interpreter from an (efficient) compiler.

- **Transformation of residual code.**

  The residual object-program produced by a staged interpreter is both a data structure that can be manipulated, and a program that can be run. Control over the form of the residual program is crucial here. The residual program is always an ML program (ML is the object language). But the user can control the form of this ML program. A goal of the translation is to make the object program use only those ML features directly supported by the target language. The restricted form of the residual object program make it possible to use the intensional analysis of object-code tools provided by MetaML to easily build the final translation step to the target language.

6.1 Benefits of the approach

This paper illustrated a step by step method for constructing correct and efficient implementations of DSLs. The method has the following advantages over building a DSL implementation in an ad-hoc fashion.

- **Simplicity.** We divide the task of DSL implementation of DSL into small manageable tasks. The compiler is constructed by a method of refinement, and we use special abstraction mechanisms so that each step addresses only a single aspect of the compiler.

- **Reuse.** Our method provides many opportunities for reuse. By using the abstraction methods of monads and staging, much of the code remains unchanged between refinement steps. In addition, monad implementations are reusable across DSLs, and multiple DLS using the same target language can reuse the intensional analysis.

- **Control.** Instead of using a fixed set of techniques or tool to generate compilers, we outline a method which provides users control over each step. A good impedance match between low-level features of the target language and the high-level DSL is necessary for good performance. Since every compiler is different, users need such fine grained control.

- **Correctness.** The METAML type system provides major support for ensuring the correctness of the compilers generated. It is simply not possible to write a type-incorrect translation. But type-correctness is not enough. We wish to prove other correctness properties as well, such as the equivalence between the artifacts produced by each step of the method. We believe that it is possible for each step to make explicit its proof obligations, and because each step produces a functional program, it is possible to use equational reasoning to prove these obligations

6.2 The Implementation

Everything you have seen in this paper, except the higher order pattern matching over code, has been implemented in the METAML implementation. The examples are actual runs of the system.

The higher order pattern matching is currently under development. We found the normalizing effect of the monad laws so compelling that we implemented them in an ad-hoc fashion inside the METAML system.

References

[1] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 308–320, New York, June 1993. ACM Press. Copenhagen.

[2] Grady Campbell. Abstraction-based reuse repositories. Technical Report REUSE-REPOSITORIES-89041-N, Software Productivity Consortium Services Corporation, 2214 Rock Hill Road, Herndon, Virginia 22070, June 1989.

[3] Grady Campbell, Stuart Faulk, and David Weiss. Introduction to Synthesis. Technical Report INTRO-SYNTHESIS-PROCESS-90019-N, Software Productivity Consortium Services Corporation, 2214 Rock Hill Road, Herndon, Virginia 22070, 1990.

[4] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 January 1996.

[5] O Danvy, J Koslowski, and K Malmkjaer. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 91.

[6] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[7] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark*, June 1993.

[8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Series editor C. A. R. Hoare. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[9] Paul Hudak Simon Peyton Jones, Philip Wadler, Brian Boutel, John Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):Section R, 1992.

[10] Peter Lee. *Realistic Compiler Generation.* Foundations of Computing Series. MIT Press, 1989.

[11] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[12] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *ESOP'96: 6th European Symposium on Programming*, number 1058 in LNCS, pages 333–343, Linkoping, Sweden, January 1996.

[13] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.

[14] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier Science Publishers B. V. (North-Holland), 1990.

[15] Peter D. Mosses. SIS-semantics implementation system, reference manual and users guide. Technical Report DAIMI report MD-30, University of Aarhus, Aarhus, Denmark, 1979.

[16] Peter D. Mosses. Action semantics. *Cambridge Tracts in Theoretical Computer Science*, (26), 1992.

[17] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *23rd ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1996.

[18] L. Paulson. *Methods and Tools for Compiler Construction, B. Lorho (editor).* Cambridge University Press, 1984.

[19] Lawrence Paulson. A sematics directed compiler generator. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 224–233. ACM, January 1982.

[20] John Peterson, Kevin Hammond, et al. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.3. Technical report, Yale University, May 1996.

[21] Frank Pfenning, Jolle Despeyroux, and Carsten Schrmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997.

[22] Frank Pfenning, Gilles Dowek, Thrse Hardin, and Claude Kirchner. Unification via explicit substitutions: The case of higher-order patterns. In *Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, Bonn, Germany, September 1996.

[23] Calton Pu and Jonathan Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.

[24] Guy Steele. Building interpreters by composing monads. In *21st Annual ACM Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.

[25] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

[26] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, Aalborg, Denmark, 13–17July 1998.

[27] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.

[28] Philip Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pages 61–78, June 1990.

[29] Philip Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pages 61–78, June 1990.

[30] Philip Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[31] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

[32] Tanya Widen. Formal language design in the context of domain engineering. Master's thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, October 1997.

[33] Tanya Widen and James Hook. Software design automation: Language design in the context of domain engenieering. In *The 10th International Conference on Software Engineering & Knowledge Engineering (SEKE'98)*, pages 308–317, San Francisco Bay, California, June 1998.

Erasure for Termination Proofs

Hongwei Xi[1] * and Joachim Steinbach[2]

[1] Department of Computer Science and Engineering
Oregon Graduate Institute
P.O. Box 91000, Portland, OR 97291, USA
e-mail: hongwei@cse.ogi.edu

[2] Institut für Informatik
Technische Universität München
80290 München, Germany
e-mail: steinbac@in.tum.de

Abstract. We introduce a technique to facilitate termination proofs for term rewriting systems. We especially focus on innermost termination. The main features of this technique lie in its simplicity and effectiveness in practice. This work can be regarded as an application of the general notion *termination through transformation* to both termination and innermost termination proofs.

1 Introduction

It is a highly significant question to determine whether a term rewriting system (TRS) is terminating. In theorem proving, TRSs are widely used for a variety of purposes. For instance, it is often desirable to transform a set of equality rules into a TRS in order to reduce the search space. Also TRSs can be used for proving the termination of both functional and logic programs.

Though termination is an undecidable property of TRSs in general, there have been many techniques developed for facilitating termination proofs. Some surveys are given in [Der87,Ste95b]. As mentioned in [MOZ96], techniques for termination proofs can be generally classified into two categories.

- Basic techniques such as various path orderings [Pla78,KL80,Der82], Knuth-Bendix ordering [KB70], and polynomial interpretations [Lan79,BL87] that apply directly to a TRS.
- Transformational approaches which in general transform a TRS into another TRS such that the termination of the latter implies that of the former and the latter can be proven terminating more easily. For instance, transformation orderings [BL90,Ste95a], semantic labelling [Zan95] and freezing [Xi98] belong to this category. Also the dependency pair approach [AG97,AG98] can be loosely classified into this category since it transforms a TRS into a set of dependency pairs.

There are also various results on modular termination, which basically give the sufficient conditions on two terminating TRSs that imply the termination of their union. The importance of modularity results is evident. It is often true that new TRSs are formed on top of existing TRSs. With modularity results, it is possible to reduce the termination of new TRSs to that of the existing ones. In this paper, we adopt a transformational approach for establishing some results on modular termination and innermost termination. Given a TRS $\mathcal{R}$, we intend to split $\mathcal{R}$ into the union of $\mathcal{R}_1$ and $\mathcal{R}_2$, and then prove that the (innermost) termination of $\mathcal{R}_1$ implies that of $\mathcal{R}$ under some conditions.

We say that a TRS is innermost terminating if there is no infinite innermost rewriting sequence in this TRS. Roughly speaking, innermost rewriting means that we can rewrite a term only if all of its proper subterms are in normal form. To some extent, innermost rewriting can model the notion of call-by-value evaluation in functional programming, though there are usually some special rules for handling

* Partially supported by the United States Air Force Materiel Command (F19 628-96-C-0161) and the Department of Defense.

conditionals. Also it is proven in [AZ95] that the innermost termination of the TRS transformed from a logic program implies the termination of the logic program. Therefore, the study on innermost termination is of significant relevance to the study of termination of functional and logic programs. Moreover, there are also various results which relates innermost termination to termination [Gra95]. This allows us to reduce termination to innermost termination for some TRSs, where the latter is often easier to prove.

We now present an example to illustrate the erasure technique before going into further details. It is frequent to encounter hierarchical combination of TRSs when we transform functional programs into TRSs. The simple reason is that defined functions are used to define new functions. For instance, the following function purge defined in ML [MTHM97] removes all duplicates from a given (integer) list while the function remove deletes all the elements equal to some given value.

```
fun remove(x, nil) = nil
  | remove(x, cons(y, ys)) =
    if x = y then remove(x, ys) else cons(y, remove(x, ys))

fun purge(nil) = nil
  | purge(cons(x, xs)) = cons(x, purge(remove(x, xs)))
```

When proving termination of such a functional program, the following aspect must be taken into consideration:

> Usually, the programmer applies a semantic argument such as a measure function in order to show that the defined function is terminating. For example, the function purge is terminating because the length of the list $remove(x, ys)$ is not greater than that of ys. Note that it is in general an exceedingly difficult task to synthesize such a measure function from the structure of a program.

The program can be transformed into the following TRS $\mathcal{R}_{\mathrm{pg}}$ [1].

$$
\begin{array}{ll}
(1) & remove(x, nil) \to nil \\
(2) & remove(x, cons(y, ys)) \to if(x = y, remove(x, ys), cons(y, remove(x, ys))) \\
(3) & purge(nil) \to nil \\
(4) & purge(cons(x, xs)) \to cons(x, purge(remove(x, xs)))
\end{array}
$$

It seems difficult to prove the termination of this TRS with a syntactic approach. We can transform this TRS into the following TRS $\mathcal{R}^1_{\mathrm{pg}}$ with the erasure technique (ET) [2].

$$
\begin{array}{rl}
(1') & nil \to nil \\
(2.1') & cons(y, ys) \to ys \\
(2.2') & cons(y, ys) \to cons(y, ys) \\
(3') & purge(nil) \to nil \\
(4') & purge(cons(x, xs)) \to cons(x, purge(xs))
\end{array}
$$

In this case, we project a term beginning with *remove* to the second argument of *remove* and a term beginning with *if* to either the second or the third argument of *if*. Under the recursive path ordering RPO with the precedence $purge \succ cons$, the rules (2.1′), (3′) and (4′) can be strictly ordered and the rules (1′) and (2.2′) can be ordered. We now informally argue that $\mathcal{R}_{\mathrm{pg}}$ is terminating. Suppose that there is an infinite innermost $\mathcal{R}_{\mathrm{pg}}$-rewriting sequence. We will show that this sequence induces an infinite $\mathcal{R}^1_{\mathrm{pg}}$-rewriting sequence. We then observe that this induced sequence cannot have infinitely many applications of those strictly ordered rules. Therefore, there is an infinite $\mathcal{R}^1_{\mathrm{pg}}$-rewriting sequence in which only applied rules are either (1′) or (2.2′). We will then prove this implies that there is an infinite innermost $\mathcal{R}_{\mathrm{pg}}$-rewriting sequence in which the only applied rules are either (1) or (2). This is a contradiction since the

[1] We omit the rules involving = and *if* at this moment.

[2] The following is slightly different from the actual application of ET for the purpose of presentation.

TRS consisting of rules (1) and (2) is easily proven to be terminating. Therefore, we conclude that $\mathcal{R}$ is innermost terminating. This argument will be substantiated in Section 3.

As already mentioned, most of the programmers use semantic arguments to prove termination. This is a powerful and flexible approach but it is also too semantic to be largely automated. On the other hand, the limited erasure technique is syntactic, and thus it is reasonable to expect that this approach can be combined with other approaches such as the freezing technique to facilitate automatic innermost termination proofs. However, we observe in practice [SX98] that it is even questionable to scale an approach as simple as RPOS, not mentioning other more involved techniques. Therefore, we expect that a more promising direction is to apply the erasure technique interactively. We shall make this point more clear with concrete examples.

This paper is organized as follows. In Section 2, we briefly explain the notations and introduce some basic concepts. We present the erasure technique (ET) for innermost termination proofs in Section 3 and establish the correctness of ET. This section constitutes the main contribution of the paper. We then mention some closely related work and conclude. We also present some examples in Appendix A, which can be of some assistance for the reader to understand the presented work if necessary.

2 Preliminaries

In general, we stick to the notations in [DJ91] though some minor modifications may occur. We briefly summarize the notations and develop some concepts needed later.

2.1 Basics

We fix a countably infinite set $\mathcal{X}$ of variables $x, y, \ldots$ and use $\mathcal{F}$ for a (finite) set of function symbols $f, g, \ldots$. Note that every function symbol f is of a fixed arity $\mathcal{A}r(f)$ and f is a constant if $\mathcal{A}r(f) = 0$. We assume that there is at least one constant in $\mathcal{F}$. Let $\mathcal{T}(\mathcal{F}, \mathcal{X})$ denote the set of terms over $\mathcal{F}$ and $\mathcal{X}$, and $\mathcal{T}(\mathcal{F})$ for the set of ground terms over $\mathcal{F}$. Given a term t, $\mathcal{V}ar(t)$ is the set of variables that occur in t. We use $l \to r$ for a rewrite rule, where we require $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. We use σ for substitutions and $\mathrm{dom}(\sigma)$ for its (finite) domain. Also $t\sigma$ stands for the result of applying σ to t.

Definition 1. Contexts C *are defined as follows.*

1. $[]$ *is a context, and*
2. $f(t_1, \ldots, t_{i-1}, C, t_{i+1}, \ldots, t_n)$ *is a context if* $\mathcal{A}r(f) = n$ *and* C *is a context.*

$C[t]$ is the term obtained from replacing the hole $[]$ in C with term t.

A TRS $\mathcal{R}$ over $\mathcal{F}$ is a set of rewrite rules over $\mathcal{T}(\mathcal{F}, \mathcal{X})$. A function symbol f is an $\mathcal{R}$-defined function if f is the root symbol of l for some rewrite rule $l \to r$ in $\mathcal{R}$, and f is a $\mathcal{R}$-constructor if it is not an $\mathcal{R}$-defined function. We often use c for constructors.

Given a TRS $\mathcal{R}$, we write $t_1 \to_{\mathcal{R}} t_2$ if $t_1 = C[l\sigma]$ and $t_2 = C[r\sigma]$ and $l \to r$ is a rewrite rule in $\mathcal{R}$, and we may also write $t_1 \to_{\mathcal{R}} t_2/\langle C, l \to r, \sigma\rangle$ to make this explicit. A term t is in $\mathcal{R}$-normal form if there exists no t' such that $t \to_{\mathcal{R}} t'$ holds. If $l \to r \in \mathcal{R}$ and all proper subterms of $l\sigma$ are in $\mathcal{R}$-normal form, we say $t_1 = C[l\sigma]$ rewrites to $t_2 = C[r\sigma]$ through *innermost* rewriting, and we use $\xrightarrow{i}_{\mathcal{R}}$ for such a rewriting relation. Also we use $t \to^{0/1} t'$ to mean that either $t = t'$ or $t \to t'$.

We use $\to^*$ for the transitive and reflexive closure of a relation $\to$. $\mathcal{R}$ is (innermost) terminating if there exists no infinite (innermost) $\mathcal{R}$-rewriting sequence. Given a substitution σ, σ is $\mathcal{R}$-normal if $\sigma(x)$ is in $\mathcal{R}$-normal form for every $x \in \mathrm{dom}(\sigma)$. The following definition is less standard.

Definition 2. *Given a term t, t is skeleton $\mathcal{R}$-normal if we always obtain terms in $\mathcal{R}$-normal form by replacing occurrences of variables in t with terms in $\mathcal{R}$-normal form. Note that we do not have to replace occurrences of the same variable with the same terms. Similarly, t is skeleton $\mathcal{R}$-terminating if we always obtain $\mathcal{R}$-terminating terms by replacing occurrences of variables in t with $\mathcal{R}$-terminating terms.*

We have the following limited method to construct skeleton $\mathcal{R}$-normal terms.

Proposition 1. *Let $\mathcal{R}$ be a TRS.*

1. *Every variable is skeleton $\mathcal{R}$-normal.*
2. *$c(t_1, \ldots, t_n)$ is skeleton $\mathcal{R}$-normal if c is an $\mathcal{R}$-constructor and t_i are skeleton $\mathcal{R}$-normal for $i = 1, \ldots, n$.*

Proof This is straightforward by the definition. ∎

In other words, $\mathcal{R}$-constructor terms, that is, terms constructed from $\mathcal{R}$-constructors and variables, are skeleton $\mathcal{R}$-normal. Similarly, $\mathcal{R}$-constructor terms are also $\mathcal{R}$-terminating.

We use the notation $\succeq$ for a quasi ordering and $\succ$ for the strict part of $\succeq$. A reduction ordering is an ordering $\succeq$ such that its strict part $\succ$ is well-founded and both $\succeq$ and $\succ$ are compatible with the term structure and stable under substitutions. One of the most well-known and widely used reduction orderings is the *recursive path ordering* RPOS *with status* [Der82,KL80]. Please see [Ste95b] for further details.

Remark 1. We say that a rewrite rule $l \to r$ is *strictly* ordered under $\succeq$ if $l \succ r$, and $l \to r$ is ordered if $l \succeq r$.

2.2 Hierarchical Combination

Definition 3. *Given two TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$, we say $\mathcal{R}_1$ and $\mathcal{R}_2$ form a* hierarchical combination *$\mathcal{R}_1 \cup \mathcal{R}_2$ if no defined function symbols in $\mathcal{R}_2$ have appearances in $\mathcal{R}_1$. Given a term t, a subterm of t is called an $\mathcal{R}_2$-subterm if the root symbol of the subterm is a $\mathcal{R}_2$-defined function symbol.*

Notice that hierarchical combination occurs naturally when we transform functional programs into TRSs: defined functions are used to define new functions.

We omit the proof of the following lemma since it is really a bit of folklore in term rewriting.

Lemma 1. *Suppose that two TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$ form a hierarchical combination $\mathcal{R}$. We have the following.*

1. *If all $\mathcal{R}_2$-subterms of t are in $\mathcal{R}$-normal form and $t \to_{\mathcal{R}} t'$, then all $\mathcal{R}_2$-subterms of t' are in $\mathcal{R}$-normal form.*
2. *If $\mathcal{R}_1$ is terminating and all $\mathcal{R}_2$-subterms of t are in $\mathcal{R}$-normal form, then t is (innermost) $\mathcal{R}$-terminating, that is, there is no infinite (innermost) $\mathcal{R}$-rewriting sequence from t.*

In the following presentation, we may omit the prefix "$\mathcal{R}$-" if it is irrelevant or it is clear from the context which $\mathcal{R}$ we refer to.

2.3 Erasure

Generally speaking, t_1 is an erasure of t_2 if t_1 can be obtained from erasing some function symbols and subterms in t_2. In other words, t_1 embeds into t_2. However, it will soon be clear that some embedding may not be erasure.

For every function symbol f in $\mathcal{F}$ with arity n, we associate with it the following rewrite rules for $i = 1, \ldots, n$.

$$(f\text{-o-}i)\ f(x_1, \ldots, x_n) \to f(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$$
$$(f\text{-p-}i)\ f(x_1, \ldots, x_n) \to x_i$$

An $(f\text{-o-}i)$ rule is called an *omitting rule* and an $(f\text{-p-}i)$ a *projection rule.* Both of these rules are called erasure rules. Notice that an $(f\text{-o-}i)$ rule changes the arity of f. Also we say that $(f\text{-p-}i)$ is not argument-dropping if $\mathcal{A}r(f) = 1$. All other erasure rules are argument-dropping.

Given a set $\mathcal{S}$ of erasure rules in which there is *at most* one rule associated with f for every $f \in \mathcal{F}$, we call $\mathcal{S}$ an erasure TRS. The $\mathcal{S}$-erasure of t is the $\mathcal{S}$-normal form of t, which is alternatively defined as follows.

Definition 4. *Given an erasure TRS $\mathcal{S}$, we use $|t|_{\mathcal{S}}$ for the $\mathcal{S}$-erasure of t and $\epsilon(t)_{\mathcal{S}}$ for the set of terms erased from t. In general, we omit the subscript $\mathcal{S}$ if there is no risk of confusion.*

$$|t| = \begin{cases} t & \text{if } t \text{ is a variable;} \\ f(|t_1|, \ldots, |t_{i-1}|, |t_{i+1}|, \ldots, |t_n|) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } (f\text{-o-}i) \in \mathcal{S}; \\ |t_i| & \text{if } t = f(t_1, \ldots, t_n) \text{ and } (f\text{-p-}i) \in \mathcal{S}; \\ f(|t_1|, \ldots, |t_n|) & \text{if } t = f(t_1, \ldots, t_n) \text{ and otherwise.} \end{cases}$$

$$\epsilon(t) = \begin{cases} \emptyset & \text{if } t \text{ is a variable;} \\ \{t_i\} \cup \bigcup_{j \in \{1,\ldots,n\} \setminus \{i\}} \epsilon(t_j) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } (f\text{-o-}i) \in \mathcal{S}; \\ \{t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\} \cup \epsilon(t_i) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } (f\text{-p-}i) \in \mathcal{S}; \\ \bigcup_{j \in \{1,\ldots,n\}} \epsilon(t_j) & \text{if } t = f(t_1, \ldots, t_n) \text{ and otherwise.} \end{cases}$$

The erasure of rule $l \to r$ is $|l| \to |r|$, and the erasure of $\mathcal{R}$ is defined analogously. Note that the erasure of a rewrite rule may not always be a legal rewrite rule. For instance, the $\mathcal{S}$-erasure of $if(false, x, y) \to y$ is $x \to y$ for $\mathcal{S} = \{(if\text{-p-}2)\}$, which is illegal. Similarly, the erasure of a TRS may not be a legal TRS.

The erasure $|C|$ of a context C can be defined in a straightforward manner. However, $|C|$ may not be a context since the hole $[]$ in C may be erased away. In this case, we write $|C|[t]$ simply for $|C|$. Given a substitution σ, its erasure $|\sigma|$ is a substitution with the same domain and $|\sigma|(x) = |\sigma(x)|$ for every $x \in \mathrm{dom}(\sigma)$.

Proposition 2. *Given a context C, a term t and a substitution σ, we have $|C[t]| = |C|[|t|]$ and $|t\sigma| = |t||\sigma|$.*

Proof This is straightforward from a structural induction on C and t, respectively. ∎

Lemma 2. *Suppose that the erasure $|\mathcal{R}|$ of a TRS $\mathcal{R}$ is also a TRS. If $t_1 \to_{\mathcal{R}} t_2$, then $|t_1| \to^{0/1}_{|\mathcal{R}|} |t_2|$.*

Proof Assume $t_1 = C[l\sigma]$ and $t_2 = C[r\sigma]$ for some σ, where $l \to r \in \mathcal{R}$. If $|C|$ is *not* a context, then $|t_1| = |C| = |t_2|$. Otherwise, $|t_1| = |C|[|l||\sigma|]$ and $|t_2| = |C|[|r||\sigma|]$ by Proposition 2. Since $|l| \to |r| \in |\mathcal{R}|$, we have $|t_1| \to_{|\mathcal{R}|} |t_2|$. Clearly, if $|C|$ is a context, then $|t_1| \to_{|\mathcal{R}|} |t_2|$. ∎

Note that for every $f \in \mathcal{F}$ with arity n, we can introduce the following omitting rule, where $1 \leq i_1 < \ldots < i_k \leq n$.

$$(f\text{-o-}(i_1, \ldots, i_k))\ \ f(x_1, \ldots, x_n) \to f(x_1, \ldots, x_{i_1-1}, x_{i_1+1}, \ldots, x_{i_k-1}, x_{i_k+1}, \ldots, x_n)$$

In other words, this rule drops the subterms of $f(t_1, \ldots, t_n)$ at the positions $i_1, \ldots, i_k$. This rule is argument-dropping. Note that this is a *single* rule, which should not be regarded as a combination of several omitting rules. Also it should be clear that all the previous results involving erasure still hold in the presence of such omitting rules.

3 Erasure for Termination Proofs

The erasure technique (ET) is mainly to facilitate modular innermost termination proofs for TRSs. Notice that innermost termination implies termination for overlay TRSs [Gra95], and therefore this can also facilitate (classical) termination proofs. We also show that ET can be directly applied to (classical) termination proofs. The essential idea behind ET is *simulation* as presented in [Xi98]. In general, ET can be regarded as an application of the notion *termination through transformation* to both termination and innermost termination proofs.

3.1 Elementary Versions of ET

In this section, we establish some elementary versions of the erasure technique.

Definition 5. *Given a TRS $\mathcal{R}$, we say that $l \to r \in \mathcal{R}$ has a conservative erasure if $|l| \to |r|$ is a legal rewrite rule and t is skeleton $\mathcal{R}$-normal for every $t \in \epsilon(r)$, that is, all subterms erased from r are skeleton $\mathcal{R}$-normal. If all the rules in $\mathcal{R}$ have conservative erasures, then we say $\mathcal{R}$ has a conservative erasure $|\mathcal{R}|$.*

The next theorem is the most elementary one among those for ET which we will formulate and prove. Nonetheless, this theorem has largely captured the essential idea behind ET.

Theorem 1. *Assume that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ has an erasure $\mathcal{R}' = \mathcal{R}'_1 \cup \mathcal{R}'_2$, where $\mathcal{R}'_i$ are the conservative erasures of $\mathcal{R}_i$ for $i = 1, 2$. Also assume that under some reduction ordering, every rule in $\mathcal{R}'_1$ can be ordered and every rule in $\mathcal{R}'_2$ can be strictly ordered, then the innermost termination of $\mathcal{R}_1$ implies the innermost termination of $\mathcal{R}$. In the case where all erasure rules are not argument-dropping, the termination of $\mathcal{R}_1$ implies that of $\mathcal{R}$.*

Proof Suppose that there exists an infinite innermost $\mathcal{R}$-rewriting sequence as follows:

$$t_1 \xrightarrow{i}_{\mathcal{R}} t_2 \xrightarrow{i}_{\mathcal{R}} \cdots \xrightarrow{i}_{\mathcal{R}} t_n \xrightarrow{i}_{\mathcal{R}} \cdots$$

where $t_i \to t_{i+1}/\langle C_i, l_i \to r_i, \sigma_i \rangle$ for some context C_i, rule $l_i \to r_i \in \mathcal{R}$ and substitution σ_i. We show that there is an infinite innermost $\mathcal{R}_1$-rewriting sequence.

Obviously, we can require that all proper subterms of t_1 be in $\mathcal{R}$-normal form since we are handling innermost rewriting. This implies that all terms in $\epsilon(t_1)$ are in $\mathcal{R}$-normal form. We now show inductively that this is true for all t_i $(i = 1, 2, \ldots)$ by analyzing the difference between $\epsilon(t_i)$ and $\epsilon(t_{i+1})$. Let $t \in \epsilon(t_{i+1})$ and we have the following.

- t is in $\epsilon(t_i)$. Then t is in $\mathcal{R}$-normal form by induction hypothesis.
- t is not in $\epsilon(t_i)$. Note $t_i = C_i[l_i\sigma_i]$ and $t_{i+1} = C_i[r_i\sigma_i]$. If t contains $r_i\sigma_i$, then there must be some s in $\epsilon(t_i)$ such that $s \to t$. This is impossible since all terms in $\epsilon(t_i)$ are in $\mathcal{R}$-normal form. Otherwise, t is dropped from $r_i\sigma_i$. This means that t either equals $s\sigma_i$ for some $s \in \epsilon(r)$ or t is a subterm of $\sigma_i(x)$ for some $x \in \mathrm{dom}(\sigma_i)$. In the latter case, t is obviously in $\mathcal{R}$-normal form since this is innermost rewriting. In the former case, t is in $\mathcal{R}$-normal form since s is skeleton $\mathcal{R}$-normal (note that $\mathcal{R}'$ is a conservative erasure of $\mathcal{R}$) and σ is an $\mathcal{R}$-normal substitution.

Therefore, for $i = 1, 2, \ldots$, all terms in $\epsilon(t_i)$ are in $\mathcal{R}$-normal form. By Lemma 2, we have the following.

$$|t_1| \to^{0/1}_{|\mathcal{R}|} |t_2| \to^{0/1}_{|\mathcal{R}|} \cdots \to_{\mathcal{R}} |t_n| \to^{0/1}_{|\mathcal{R}|} \cdots$$

We now show that every $\to^{0/1}_{|\mathcal{R}|}$ step in this sequence is actually a $\to_{|\mathcal{R}|}$ step. It suffices to show that $|C_i|$ is always a context for $i = 1, 2, \ldots$. Suppose that $|C_i|$ is not a context. Then $l_i\sigma_i$ is a subterm of some term in $\epsilon(t_i)$. This is impossible since all terms in $\epsilon(t_i)$ are in $\mathcal{R}$-normal form. This implies that we actually have the following.

$$|t_1| \to_{|\mathcal{R}|} |t_2| \to_{|\mathcal{R}|} \cdots \to_{\mathcal{R}} |t_n| \to_{|\mathcal{R}|} \cdots$$

Since all rules in $\mathcal{R}'_1$ are ordered and all rules in $\mathcal{R}'_2$ are strictly ordered, there must be an n such that all the rules applied after $|t_n|$ are in $\mathcal{R}'_1$. This implies that all the rules applied in the infinite innermost $\mathcal{R}$-rewriting sequence after t_n are in $\mathcal{R}_1$, that is, we have an infinite innermost $\mathcal{R}_1$-rewriting sequence. Therefore, the innermost termination of $\mathcal{R}_1$ implies that of $\mathcal{R}$.

We now prove the second part of the theorem. Suppose that all the erasure rules are not argument-dropping. Then $|C|$ is a context for every context C. Therefore, $t_1 \to_{\mathcal{R}} t_2$ implies $|t_1| \to_{|\mathcal{R}|} |t_2|$ for every pair of terms t_1 and t_2. With the same argument as before, we can show that an infinite $\mathcal{R}$-rewriting sequence induces an infinite $\mathcal{R}_1$-rewriting sequence. Therefore, the termination of $\mathcal{R}_1$ implies that of $\mathcal{R}$. ■

Notice that we assume *no* relation between $\mathcal{R}_1$ and $\mathcal{R}_2$ in Theorem 1. This is an attractive feature in practice. Suppose that we intend to prove the termination of $\mathcal{R}$. We proceed to find a conservative erasure $\mathcal{R}'$ of $\mathcal{R}$ such that all rules in $\mathcal{R}'$ can be ordered under some reduction ordering. If there are rules in $\mathcal{R}'$ which can be strictly ordered, we remove them and use $\mathcal{R}'_1$ for the set of remaining rules. We can then find $\mathcal{R}_1 \subseteq \mathcal{R}$ such that $\mathcal{R}'_1$ is the conservative erasure of $\mathcal{R}_1$. In this way, we have reduced the innermost termination of $\mathcal{R}$ to that of $\mathcal{R}_1$. If $\mathcal{R}_1$ is empty, then we have proven that $\mathcal{R}$ is innermost terminating. Clearly, there is no need for splitting $\mathcal{R}$ before applying Theorem 1.

The following TRS $\mathcal{R}_{\mathrm{wt}}$ is taken from [AG98]. Note that m, n are variables, :: is the infix operator for *cons*, [] for *nil* and $[n]$ for $cons(n, nil)$. The function *weight* computes a weighted sum of natural numbers: $weight(n_0 :: n_1 :: \cdots :: n_k :: nil) = n_0 + \Sigma_{i=1}^{k} i * n_i$.

$$
\begin{array}{ll}
(1) & sum(s(m) :: x, n :: y) \to sum(m :: x, s(n) :: y) \\
(2) & sum(0 :: x, y) \to sum(x, y) \\
(3) & sum([], y) \to y \\
(4) & weight([n]) \to n \\
(5) & weight(m :: n :: x) \to weight(sum(m :: n :: x, 0 :: x))
\end{array}
$$

The last rule is self-embedding, and therefore the TRS cannot be proven terminating with a simplification ordering. Intuitively, $\mathcal{R}_{\mathrm{wt}}$ is terminating because the length of $sum(m :: n :: x, 0 :: x)$ is less than that of $m :: n :: x$. We can use the erasure TRS $\mathcal{S} = \{(\text{sum-p-2}), (\text{s-p-1})\}$ to capture this. The following TRS $\mathcal{R}'_{\mathrm{wt}}$ is the $\mathcal{S}$-erasure of $\mathcal{R}_{\mathrm{wt}}$.

$$
\begin{array}{ll}
(1') & n :: y \to n :: y \\
(2') & y \to y \\
(3') & y \to y \\
(4') & weight([n]) \to n \\
(5') & weight(m :: n :: x) \to weight(0 :: x)
\end{array}
$$

Notice that this is a conservative erasure. For instance, let r be the right-hand side of rule (5), then $\epsilon(r)_{\mathcal{S}}$ is $\{m :: n :: x\}$, in which the term is skeleton $\mathcal{R}_{\mathrm{wt}}$-normal. Clearly, $\mathcal{R}'_{\mathrm{wt}}$ can be ordered under a RPO. Since the rules (4') and (5') are strictly ordered, we delete them. Therefore, the innermost termination of $\mathcal{R}_{\mathrm{sum}}$, which consists of the rules (1), (2) and (3), implies that of $\mathcal{R}_{\mathrm{wt}}$ by Theorem 1. The termination of $\mathcal{R}_{\mathrm{sum}}$ is readily proven with a RPOS, and thus $\mathcal{R}_{\mathrm{wt}}$ is innermost terminating. In this case $\mathcal{R}_{\mathrm{wt}}$ is terminating since it is an overlay (actually non-overlapping) TRS.

On the other hand, if we can split a TRS into some hierarchical combination, then we can take advantage of Theorem 2 below, which is a generalized version of Theorem 1. We first present a definition very close to Definition 5.

Definition 6. *Let $\mathcal{R}$ be the hierarchical combination of $\mathcal{R}_1$ and $\mathcal{R}_2$. We say that $l \to r \in \mathcal{R}$ has an $\mathcal{R}_2$-conservative erasure if $|l| \to |r|$ is a legal rewrite rule and t is skeleton $\mathcal{R}_2$-normal for every $t \in \epsilon(r)$. If all rules in $\mathcal{R}$ have $\mathcal{R}_2$-conservative erasures, then we say $\mathcal{R}$ has an $\mathcal{R}_2$-conservative erasure $|\mathcal{R}|$.*

Theorem 2. *Let $\mathcal{S}$ be an erasure TRS and $\mathcal{R}$ be the hierarchical combination of $\mathcal{R}_1$ and $\mathcal{R}_2 = \mathcal{R}_{21} \cup \mathcal{R}_{22}$ such that $|\mathcal{R}_1|$ and $|\mathcal{R}_2|$ are $\mathcal{R}_2$-conservative. Assume that under some reduction ordering, the erasure of every rule in $\mathcal{R}_1$ and $\mathcal{R}_{21}$ can be ordered and the erasure of every rule in $\mathcal{R}_{22}$ can be strictly ordered, then the innermost termination of $\mathcal{R}_1 \cup \mathcal{R}_{21}$ implies the innermost termination of $\mathcal{R}$. In the case where all erasure rules in $\mathcal{S}$ are not argument-dropping, the termination of $\mathcal{R}_1 \cup \mathcal{R}_{21}$ implies the termination of $\mathcal{R}$.*

Proof This is very similar to the proof of Theorem 1. Suppose that there exists an infinite innermost $\mathcal{R}$-rewriting sequence as follows:

$$t_1 \xrightarrow{i}_{\mathcal{R}} t_2 \xrightarrow{i}_{\mathcal{R}} \cdots \xrightarrow{i}_{\mathcal{R}} t_n \xrightarrow{i}_{\mathcal{R}} \cdots$$

where $t_i \to t_{i+1}/\langle C_i, l_i \to r_i, \sigma_i\rangle$ for some context C_i, rule $l_i \to r_i \in \mathcal{R}$ and substitution σ_i. We show that there is an infinite innermost $\mathcal{R}_1$-rewriting sequence.

Obviously, we can require that all proper subterms of t_1 be in $\mathcal{R}$-normal form since we are handling innermost rewriting. This implies that all terms in $\epsilon(t_1)$ are in $\mathcal{R}_2$-normal form. We now show inductively that this is true for all t_i $(i = 1, 2, \ldots)$ by analyzing the difference between $\epsilon(t_i)$ and $\epsilon(t_{i+1})$. Let $t \in \epsilon(t_{i+1})$ and we have the following.

- t is in $\epsilon(t_i)$. Then t is in $\mathcal{R}_2$-normal form by induction hypothesis.
- t is not in $\epsilon(t_i)$. Note $t_i = C_i[l_i\sigma_i]$ and $t_{i+1} = C_i[r_i\sigma_i]$. If t contains $r_i\sigma_i$, then there must be some s in $\epsilon(t_i)$ such that $s \to t$. Note $l_i \to r_i$ cannot be in $\mathcal{R}_2$ since s must be in $\mathcal{R}_2$-normal form. Thus, t is in $\mathcal{R}_2$-normal form by Lemma 1. Otherwise, t is dropped from $r_i\sigma_i$. This means that t either equals $s\sigma_i$ for some $s \in \epsilon(r)$ or t is a subterm of $\sigma_i(x)$ for some $x \in \mathrm{dom}(\sigma_i)$. In the latter case, t is obviously in $\mathcal{R}_2$-normal form since this is innermost rewriting. In the former case, t is in $\mathcal{R}_2$-normal form since s is skeleton $\mathcal{R}_2$-normal (note that $\mathcal{R}'$ is a $\mathcal{R}_2$-conservative erasure of $\mathcal{R}$) and σ is an $\mathcal{R}_2$-normal (actually $\mathcal{R}$-normal) substitution.

Therefore, for $i = 1, 2, \ldots$, all terms in $\epsilon(t_i)$ are in $\mathcal{R}_2$-normal form. If $l_i \to r_i \in \mathcal{R}_2$, then $|C_i|$ must be a context since $l_i\sigma_i$ would be a subterm of some $t \in \epsilon(t_i)$ otherwise, which contradicts that all terms in $\epsilon(t_i)$ are $\mathcal{R}_2$-normal. Thus, if $t_i \to_{\mathcal{R}_2} t_2$ then $|t_1| \to_{|\mathcal{R}_2|} |t_2|$. Since all rules in $|\mathcal{R}_{22}|$ are strictly ordered under some reduction ordering and all rules in $|\mathcal{R}_1| \cup |\mathcal{R}_{21}|$ are ordered, there must be a n such that for all $i > n$, $l_i \to r_i \notin \mathcal{R}_{22}$. This implies that we have an infinite innermost $\mathcal{R}$-rewriting sequence in which all applied rules are either from $\mathcal{R}_1$ or $\mathcal{R}_{21}$. Contrapositively, the innermost termination of $\mathcal{R}_1 \cup \mathcal{R}_{21}$ implies that of $\mathcal{R}$.

The second part of this theorem is really the same as that of Theorem 1. We thus omit the details. ■

We now present an application of Theorem 2. The following example is taken from the technical report version of [AG97].

Let R_1 be a TRS consisting of the following rules,

$$\begin{array}{rr} le(0,y) \to true & pred(s(x)) \to x \\ le(s(x),0) \to false & minus(x,0) \to x \\ le(s(x),s(y)) \to le(x,y) & minus(x,s(y)) \to pred(minus(x,y)) \end{array}$$

and $\mathcal{R}_2$ be a TRS consisting of the following rules.

$$\begin{array}{ll} (1) & gcd(0,y) \to 0 \\ (2) & gcd(s(x),0) \to 0 \\ (3) & gcd(s(x),s(y)) \to ifgcd(le(y,x),s(x),s(y)) \\ (4) & ifgcd(true,s(x),s(y)) \to gcd(minus(x,y),s(y)) \\ (5) & ifgcd(false,s(x),s(y)) \to gcd(minus(y,x),s(x)) \end{array}$$

Let $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. $\mathcal{R}$ is clearly a hierarchical combination of $\mathcal{R}_1$ and $\mathcal{R}_2$. We form a $\mathcal{S}$-erasure $\mathcal{R}'$ of $\mathcal{R}$ as follows for $\mathcal{S} = \{(\mathrm{pred}\text{-}p-1), (\mathrm{minus}\text{-}p-1), (ifgcd\text{-}o-1)\}$. $\mathcal{R}' = \mathcal{R}'_1 \cup \mathcal{R}'_2$, where $\mathcal{R}'_1$ consists of the following rules

$$\begin{array}{rr} le(0,y) \to true & s(x) \to x \\ le(s(x),0) \to false & x \to x \\ le(s(x),s(y)) \to le(x,y) & x \to x \end{array}$$

and $\mathcal{R}'_2$ consists of the following rules.

$$\begin{array}{ll} (1') & gcd(0,y) \to 0 \\ (2') & gcd(s(x),0) \to 0 \\ (3') & gcd(s(x),s(y)) \to ifgcd(s(x),s(y)) \\ (4') & ifgcd(s(x),s(y)) \to gcd(x,s(y)) \\ (5') & ifgcd(s(x),s(y)) \to gcd(y,s(x)) \end{array}$$

It can be readily verified that $\mathcal{R}'$ is a $\mathcal{R}_2$-conservative erasure of $\mathcal{R}$. Under the RPO with the precedence relation $gcd \approx ifgcd$ and $le \succ true, false$, all the rules in $\mathcal{R}_1'$ and the rule $(3')$ can be ordered, and the rules $(1'), (2'), (4')$ and $(5')$ can be strictly ordered. By Theorem 2, the innermost termination of $\mathcal{R}_1 \cup \{(3)\}$ implies that of $\mathcal{R}$. Since $\mathcal{R}_1 \cup \{(3)\}$ can be easily proven terminating with a RPOS, $\mathcal{R}$ is innermost termination. Note that $\mathcal{R}$ is a non-overlapping TRS and thus $\mathcal{R}$ is terminating.

In practice, we may encounter the case where $\mathcal{R}_1 = \emptyset$ when we apply Theorem 1, or $\mathcal{R}_{21} = \emptyset$ when we apply Theorem 2. Let use consider a concrete example. The TRS $\mathcal{R}$ consist of the following single rule.

$$f(g(x)) \to g(f(f(x)))$$

If we form the $\mathcal{S}$-erasure of $\mathcal{R}$ for $\mathcal{S} = \{(f\text{-p-}1)\}$, we obtain the following TRS $|\mathcal{R}|$,

$$g(x) \to g(x)$$

which cannot be strictly ordered under any reduction ordering. Therefore, if we apply Theorem 1, we make no progress. However, we can argue that $\mathcal{R}$ is terminating as follows. Suppose that there is an infinite $\mathcal{R}$-rewriting sequence:

$$t_1 \to_{\mathcal{R}} t_2 \cdots \to_{\mathcal{R}} t_n \to_{\mathcal{R}} \cdots$$

We can choose t_1 such that all proper subterms of t_1 are $\mathcal{R}$-terminating and t is $\mathcal{R}$-terminating for every t if $|t|$ is a subterm of t_1. Then t_1 must be of form $f(s)$. Since s is $\mathcal{R}$-terminating, there is some $t_n = f(g(s'))$ such that $s \to_{\mathcal{R}}^* g(s')$ and $t_{n+1} = g(f(f(s')))$. It is clear that $|t_1| = |t_{n+1}| = g(|s'|)$. Given the property of t_1, we know that s' is $\mathcal{R}$-terminating. This implies that t_{n+1} is $\mathcal{R}$-terminating, contradicting that the above $\mathcal{R}$-rewriting sequence is infinite. Therefore, there exists no infinite $\mathcal{R}$-rewriting sequence, that is, $\mathcal{R}$ is terminating. We present a formalization of this idea as follows.

Definition 7. *Let $\mathcal{S}$ be an erasure TRS. Given a simplification ordering $\succeq_1$ on terms and a quasi precedence relation $\succeq$ on a finite set of function symbols, we can define a (strict) ordering $\succ_2$ as follows. Given s and t, $s \succ_2 t$ if either $|s| \succ_1 |t|$, or $|s| \succeq_1 |t|$ and s and t are of form $f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$, respectively, and $f \succ g$ and*

- *there is no erasure rule in $\mathcal{S}$ is associated with g, or*
- *the erasure rule in $\mathcal{S}$ associated with g is an omitting rule, or*
- *$(g\text{-p-}i) \in \mathcal{S}$ and $s \succ_2 t_i$.*

Lemma 3. *The ordering $\succ_2$ defined in Definition 7 is well-founded and stable under substitutions.*

Proof This is straightforward since $\succeq_1$ is well-founded and stable under substitutions and $\succeq$ is well-founded.

■

Theorem 3. *Let $\mathcal{S}$ be an erasure TRS and $\mathcal{R}$ be the hierarchical combination of $\mathcal{R}_1$ and $\mathcal{R}_2$ such that $|\mathcal{R}_1|$ and $|\mathcal{R}_2|$ are $\mathcal{R}_2$-conservative, and the erasure of every rule in $\mathcal{R}_1$ and $\mathcal{R}_2$ can be ordered under some simplification ordering $\succeq_1$. Let $\succeq$ be a quasi precedence relation on a finite set of function symbols, and we form an ordering $\succ_2$ as described in Definition 7. Assume that for every rule $l \to r \in \mathcal{R}_2$, either r is skeleton $\mathcal{R}_2$-normal or $l \succ_2 r$. Then the innermost termination of $\mathcal{R}_1$ implies that of $\mathcal{R}_2$. If all the erasure rules in $\mathcal{S}$ are not argument-dropping and for every rule $l \to r \in \mathcal{R}_2$, either r is skeleton $\mathcal{R}_2$-terminating or $l \succ_2 r$, then the termination of $\mathcal{R}_1$ implies that of $\mathcal{R}_2$.*

Proof Assume that $\mathcal{R}_1$ is innermost terminating but $\mathcal{R}$ is not. Let $P(s)$ be a property on terms stating that s is not $\mathcal{R}$-terminating but all proper subterms of t are $\mathcal{R}$-terminating. Since $\succ_2$ is well-founded by Lemma 3, we can choose a term s such that $P(s)$ holds but $P(t)$ fails for every t satisfying $s \succ_2 t$. We can prove by a structural induction the claim that t is $\mathcal{R}$-terminating for every t such that all terms in

$\epsilon(t)$ are in $\mathcal{R}_2$ normal form and $s \succ_2 t$. Please see the proof of Theorem 4 for details. Since $P(s)$ holds, there exists an infinite innermost $\mathcal{R}$-rewriting sequence of the following form,

$$s = f(s_1, \ldots, s_m) \to_{\mathcal{R}}^* f(s'_1, \ldots, s'_m) = s' \to_{\mathcal{R}} s'' \to_{\mathcal{R}} \cdots$$

where $s_i \to_{\mathcal{R}}^* s'_i$ and s'_i are in $\mathcal{R}$-normal form for $i = 1, \ldots, m$ and $s' = l\sigma$ and $s'' = r\sigma$ and $l \to r \in \mathcal{R}$. $l \to r$ must be a rule in $\mathcal{R}_2$ by Lemma 1 (2) and r clearly cannot be skeleton $\mathcal{R}_2$-normal. Therefore, $s' \succ_2 s''$. It can be readily proven that $s \succ_2 s''$ since all rules in $\mathcal{R}$ are ordered under $\succeq_1$. Now let us assume that s'' is of form $g(t_1, \ldots, t_n)$. We do a case analysis on the form of $|s''|$.

- There exists no rule in $\mathcal{S}$ associated with g. This case is the same as the next one.
- (g-o-$(i_1, \ldots, i_k)$) $\in \mathcal{S}$. Then we have

  $$|s''| = g(|t_1|, \ldots, |t_{i_1-1}|, |t_{i_1+1}|, \ldots, |t_{i_k-1}|, |t_{i_k+1}|, \ldots, |t_n|).$$

  Note that all terms in $\epsilon(s'')$ are in $\mathcal{R}_2$-normal form since $|\mathcal{R}|$ are $\mathcal{R}_2$-conservative. We have $|s| \succeq_1 |s''| \succ_1 |t_j|$ for $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$ since $\succeq_1$ is a simplification ordering. Hence, $s \succ_2 t_j$. With the above claim, these t_j are $\mathcal{R}$-terminating since $\epsilon(t_j) \subseteq \epsilon(s'')$. Clearly, $t_{i_1}, \ldots, t_{i_k}$ are $\mathcal{R}$-terminating and this implies that all proper subterms of s'' are $\mathcal{R}$-terminating. Hence s'' is $\mathcal{R}$-terminating since $P(s'')$ holds and $s \succ_2 s''$. We have thus reached a contradiction.
- (g-p-i) $\in \mathcal{S}$. Then $|s''| = |t_i|$. We have $s' \succ_2 t_i$ by the definition of $\succ_2$, and this can lead to $s \succ_2 t_i$. With the above claim, t_i is $\mathcal{R}$-terminating since $\epsilon(t_i) \subset \epsilon(s'')$. Clearly, t_j are $\mathcal{R}$-terminating for all $j \in \{1, \ldots, i-1, i+1, n\}$. Again, this implies that s'' is $\mathcal{R}$ terminating since $P(s'')$ holds and $s \succ_2 s''$. This is a contradiction. terminating.

Therefore, $\mathcal{R}$ must be terminating. It should be straightforward to prove the second part of the theorem. ■

We present an application of Theorem 3. The following TRS $\mathcal{R}$ is due to Dershowitz.

$$\begin{array}{ll} (1) & \neg(\neg(x)) \to x \\ (2) & \neg(x \wedge y) \to \neg(\neg(\neg(x))) \vee \neg(\neg(\neg(y))) \\ (3) & \neg(x \vee y) \to \neg(\neg(\neg(x))) \wedge \neg(\neg(\neg(y))) \end{array}$$

The following is the $\mathcal{S}$-erasure $|\mathcal{R}|$ of $\mathcal{R}$ for $\mathcal{S} = \{(\neg\text{-p-}1)\}$.

$$\begin{array}{c} x \to x \\ x \wedge y \to x \vee y \\ x \vee y \to x \wedge y \end{array}$$

Clearly, all rules in $|\mathcal{R}|$ are ordered in the RPO with the precedence $\wedge \approx \vee$. Let $\succeq_1$ denote this RPO. We can form an ordering $\succ_2$ with the precedence relation $\neg \succ \wedge, \vee$ as described in Definition 7. Notice that the right side of (1) is $\mathcal{R}$-skeleton terminating and both rules (2) and (3) can be ordered under $\succ_2$. By Theorem 3, $\mathcal{R}$ is terminating since the rule in $\mathcal{S}$ is not argument-dropping.

Please see Example 6 for a more sophisticated application of Theorem 3

3.2 Nondeterministic Erasure Rules

For those who are familiar with the dependency pair approach (DPA) [AG97,AG98], it should be clear that the erasure technique presented so far can be regarded as a closely related idea recast into the framework of *termination through transformation.* However, the following development significantly separates ET from DPA.

Let us now take a look at a limitation of the erasure technique developed so far before proceeding to formulate more sophisticated versions of ET. The rules associated with *if* are the following.

$$\mathit{if}(\mathit{true}, x, y) \to x \qquad \mathit{if}(\mathit{false}, x, y) \to y$$

For the example $\mathcal{R}_{\mathrm{pg}}$, we would like to use the erasure TRS $\mathcal{S} = \{(remove\text{-p-}2), (if\text{-p-}3)\}$ so that we can erase the following rule into $cons(y, ys) \to cons(y, ys)$.

$$remove(x, cons(y, ys)) \to if(x = y, remove(x, ys), cons(x, remove(y, ys))$$

Unfortunately, we also obtain the erasure $y \to x$ for the rule $if(true, x, y) \to x$, which is not a legal rewrite rule. This is a severe limitation in practice since *if* is widely used in defining TRSs. We extend the definition of erasure to resolve this problem.

Definition 8. *Given a function symbol f with arity n and $1 \le i_i < \cdots < i_k \le n$, the following nondeterministic rule is also an erasure rule.*

$$(f\text{-p-}(i_1, \ldots, i_k))\ f(x_1, \ldots, x_n) \to \{x_{i_1}, \ldots, x_{i_k}\}$$

This means that $f(x_1, \ldots, x_n)$ can rewrite to x_{i_j} for each $1 \le j \le k$. This rule is not argument-dropping if $\{i_1, \ldots, i_k\} = \{1, \ldots, n\}$.

With this extension, the erasure $|t|$ of a term t is a multiset of terms, which can be defined as follows.

$$|t| = \begin{cases} \{t\} & \text{if } t \text{ is a variable;} \\ f(|t_1|, \ldots, |t_{i_i-1}|, |t_{i_i+1}|, \ldots, |t_{i_k-1}|, |t_{i_k+1}|, \ldots, |t_n|) & \\ & \text{if } t = f(t_1, \ldots, t_n) \text{ and } (f\text{-o-}(i_1, \ldots, i_k)) \in \mathcal{S}; \\ |t_{i_1}| \cup \cdots \cup |t_{i_k}| & \text{if } t = f(t_1, \ldots, t_n) \text{ and } (f\text{-p-}(i_1, \ldots, i_k)) \in \mathcal{S}; \\ f(|t_1|, \ldots, |t_n|) & \text{if } t = f(t_1, \ldots, t_n) \text{ and otherwise.} \end{cases}$$

We use the notation $f(|t_1|, \ldots, |t_n|)$ for the multiset

$$\{f(s_1, \ldots, s_n) \mid s_i \in |t_i| \text{ for } 1 \le i \le n\},$$

that is, the multiset of terms $f(s_1, \ldots, s_n)$, where s_i range over $|t_i|$ for $1 \le i \le n$. We also present the definition for $\epsilon(t)$, which is the set of terms erased from t.

$$\epsilon(t) = \begin{cases} \emptyset & \text{if } t \text{ is a variable;} \\ \{t_{i_1}, \ldots, t_{i_k}\} \cup \bigcup_{j \in \{1,\ldots,n\} \setminus \{i_1,\ldots,i_k\}} \epsilon(t_j) & \\ & \text{if } t = f(t_1, \ldots, t_n) \text{ and } (f\text{-o-}(i_1, \ldots, i_k)) \in \mathcal{S}; \\ \{t_1, \ldots, t_{i_1-1}, t_{i_1+1}, \ldots, t_{i_k-1}, t_{i_k+1}, \ldots, t_n\} \cup \epsilon(t_{i_1}) \cup \cdots \cup \epsilon(t_{i_k}) & \\ & \text{if } t = f(t_1, \ldots, t_n) \text{ and } (f\text{-p-}(i_1, \ldots, i_k)) \in \mathcal{S}; \\ \bigcup_{j \in \{1,\ldots,n\}} \epsilon(t_j) & \text{if } t = f(t_1, \ldots, t_n) \text{ and otherwise.} \end{cases}$$

In addition, the erasure $|C|$ of context C is a multiset, in which every element is either a context or a term. The erasure $|\sigma|$ of substitution σ with a finite domain is defined below.

$$|\sigma| = \{\tau \mid \mathrm{dom}(\tau) = \mathrm{dom}(\sigma) \text{ and } \tau(x) \in |\sigma(x)| \text{ for every } x \in \mathrm{dom}(\tau)\}$$

Definition 9. *Let $\succeq$ be an ordering on terms. We extend this ordering to the (nonempty) multisets of terms as follows: $S \succeq^{max} (\succ^{max}) T$ if and only if for every $t \in T$ there is an $s \in S$ such that $s \succeq (\succ)\ t$, where S and T stand for the multisets of terms.*

Please notice the difference between $\succeq^{max}$ and $\succeq^{\textcircled{m}}$. For instance, we have $\{c(x)\} \succeq^{max} \{x, c(x)\}$ but $\{c(x)\} \not\succeq^{\textcircled{m}} \{x, c(x)\}$. Also we observe that $\succ^{max}$ is well-founded on the multisets of terms if $\succ$ is well-founded on terms.

Given a rule $l \to r$, the erasure of this rule is $|l| \to |r|$. The erasure of a TRS is defined similarly. Note that we no longer consider the erasure of a rule (TRS) as a rule (TRS), but refer it as a *rule (TRS) erasure*. Given a reduction ordering $\succeq$ on terms, we say that the rule erasure $|l| \to |r|$ is strictly ordered under $\succeq$ if $|l| \succ^{max} |r|$, and it is ordered if $|l| \succeq^{max} |r|$.

For instance, for $\mathcal{S} = \{(remove\text{-p-}2), (if\text{-p-}(2,3))\}$, the $\mathcal{S}$-erasure of $\mathcal{R}_{\mathrm{pg}}$ is the following. We write a term for the singleton set consisting of the term to support transparent syntax.

$$\begin{array}{llll} (1') & nil \to nil & (2') & cons(y, ys) \to \{ys, cons(y, ys)\} \\ (3') & purge(nil) \to nil & (4') & purge(cons(x, xs)) \to cons(x, purge(xs)) \end{array}$$

Under the RPO with the precedence $purge \succ cons$, the top two rule erasures are ordered and the rest are strictly ordered.

Definition 10. *An ordering $\succeq$ is a weak reduction ordering if its strict part $\succ$ is well-founded and stable under substitutions and $\succeq$ is compatible wrt. term structure and stable under substitutions. Notice that a weak reduction ordering $\succeq$ may not be a reduction ordering since it is not required that $\succ$ be also compatible wrt. term structure.*

Lemma 4. *Let $\succeq$ be a weak reduction ordering which is total on ground terms. Given a ground substitution σ, that is, $\sigma(x)$ is a ground term for every $x \in \mathrm{dom}(\sigma)$, we have the following for every erasure TRS $\mathcal{S}$.*

1. *There exists a substitution $\sigma_{max} \in |\sigma|_{\mathcal{S}}$ such that for every $\tau \in |\sigma|_{\mathcal{S}}$, $\sigma_{max}(x) \succeq \tau(x)$ hold for all $x \in \mathrm{dom}(\sigma)$.*
2. *If t is a term such that $\mathcal{V}ar(t) \subset \mathrm{dom}(\sigma)$, then for every $s_2 \in |t\sigma|_{\mathcal{S}}$ there exists $s_1 \in |t|_{\mathcal{S}}$ such that $s_1\sigma_{max} \succeq s_2$.*

Proof For every $x \in \mathrm{dom}(\sigma)$, we can choose a term $t_x \in |\sigma(x)|$ such that $t_x \succeq t$ for all $t \in |\sigma(x)|$ since $\succeq$ is total on ground terms. Let σ_{max} be the substitution with domain $\mathrm{dom}(\sigma)$ and $\sigma_{max}(x) = t_x$ for all $x \in \mathrm{dom}(\sigma)$. By the definition of $|\sigma|$, we obtain (1). (2) can be readily proven by a structural induction on t. ■

Notice that we actually only require that the weak reduction ordering $\succeq$ be extendable to a total ordering on ground terms. For example, reduction orderings based on RPOS or polynomial interpretations satisfy the requirement.

Definition 11. *Let $\mathcal{S}$ be an erasure TRS. For every weak reduction ordering $\succeq$ which is total on ground terms, we can define an ordering $\succeq_{\mathcal{S}}^{max}$ as follows.*

$$t_1 \succeq_{\mathcal{S}}^{max} t_2 \text{ if and only if } |t_1|_{\mathcal{S}} \succeq^{max} |t_2|_{\mathcal{S}}$$

The next proposition states a crucial property of $\succeq_{\mathcal{S}}^{max}$.

Proposition 3. *Given a weak reduction ordering $\succeq$ and an erasure TRS $\mathcal{S}$, the ordering $\succeq_{\mathcal{S}}^{max}$ on terms is also a weak reduction ordering.*

Proof By Lemma 4, it is straightforward to prove that both $\succ^{max}$ and $\succeq_{\mathcal{S}}^{max}$ are stable under substitutions. The compatibility of $\succeq_{\mathcal{S}}^{max}$ with term structure follows from the definition of the erasure function $|\cdot|_{\mathcal{S}}$. ■

In general, it does not hold that $t_1 \succ_{\mathcal{S}}^{max} t_2$ implies $C[t_1] \succ_{\mathcal{S}}^{max} C[t_2]$ for every context C even if $\succeq$ is a reduction ordering. Therefore, we cannot infer that $\succeq_{\mathcal{S}}^{max}$ is a reduction ordering under the assumption that $\succeq$ is.

Theorem 4. *Let TRS $\mathcal{R}$ be the hierarchical combination of $\mathcal{R}_1$ and $\mathcal{R}_2$ and $|\mathcal{R}_1|_{\mathcal{S}}$ and $|\mathcal{R}_2|_{\mathcal{S}}$ are $\mathcal{R}_2$-conservative TRS erasures for some erasure TRS $\mathcal{S}$. Assume that $\succeq$ is a weak reduction ordering which is total on ground terms and $l \succeq_{\mathcal{S}}^{max} r$ for every rule $l \to r \in \mathcal{R}_1$ and $l \succ_{\mathcal{S}}^{max} r$ for every rule $l \to r \in \mathcal{R}_2$. Then the innermost termination of $\mathcal{R}_1$ implies that of $\mathcal{R}$.*

Proof Assume that $\mathcal{R}_1$ is innermost terminating but $\mathcal{R}$ is not. Let $P(t)$ be a property on terms stating that t is not innermost $\mathcal{R}$-terminating and every proper subterm of t is innermost terminating. We can choose a ground term t such that $P(t)$ holds and $P(s)$ fails for every term s satisfying $t \succ_{\mathcal{S}}^{max} s$ since $\succ_{\mathcal{S}}^{max}$ is well-founded. We now prove by a structural induction on s that s is innermost terminating if $t \succ_{\mathcal{S}}^{max} s$ and all terms in $\epsilon(s)_{\mathcal{S}}$ are innermost terminating. Assume that s is of form $f(s_1, \ldots, s_n)$. We do a case analysis on the form of $|s|$.

- No erasure rule in $\mathcal{S}$ is associated with f. This case is the same as the next one.
- The erasure rule (f-o-$(i_1, \ldots, i_k)$) is in $\mathcal{S}$. Then

$$|s| = f(|s_1|, \ldots, |s_{i_1-1}|, |s_{i_1+1}|, \ldots, |s_{i_k-1}|, |s_{i_k+1}|, \ldots, |s_n|).$$

  $\succeq$ must be a simplification ordering on ground terms since it is total on them. Therefore, for every $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$, $t \succ_{\mathcal{S}}^{max} s \succeq_{\mathcal{S}}^{max} s_j$, and thus s_j is innermost terminating since $\epsilon(s_j) \subset \epsilon(s)$ implies that all terms in $\epsilon(s_j)$ are innermost terminating. Also s_{i_j} are innermost terminating for $1 \leq j \leq k$ since they are in $\epsilon(s)$. Therefore, all proper subterms of s are innermost terminating. Given the property of t, s is innermost $\mathcal{R}$-terminating.
- The erasure rule (f-p-$(i_1, \ldots, i_k)$) is in $\mathcal{S}$. This is similar to the previous case.

Thus we have proven the claim that s is innermost $\mathcal{R}$-terminating if $t \succ_{\mathcal{S}}^{max} s$ and all terms in $\epsilon(s)$ are innermost terminating.

Assume that t is of form $f(t_1, \ldots, t_n)$. Since t is not innermost $\mathcal{R}$-terminating and all proper terms of t are innermost $\mathcal{R}$-terminating, there is an infinite innermost rewriting sequence beginning with the following form,

$$t = f(t_1, \ldots, t_n) \overset{i}{\rightarrow}{}^*_{\mathcal{R}} f(t'_1, \ldots, t'_n) = t' \overset{i}{\rightarrow}_{\mathcal{R}} t''$$

where $t_i \overset{i}{\rightarrow}{}^*_{\mathcal{R}} t'_i$ and t'_i are in $\mathcal{R}$-normal form for $1 \leq i \leq n$, and $t' = l\sigma$ and $t'' = r\sigma$ for some $l \rightarrow r \in \mathcal{R}$. Note that f cannot be a defined function symbol in $\mathcal{R}_1$ by Lemma 1 (2). Hence, $l \rightarrow r \in \mathcal{R}_2$, and this implies $l \succ_{\mathcal{S}}^{max} r$. Therefore, we have $t \succeq_{\mathcal{S}}^{max} t' \succ_{\mathcal{S}}^{max} t''$. Note that all terms in $\epsilon(t'')$ are $\mathcal{R}_2$-normal since all terms in $\epsilon(r)$ are skeleton $\mathcal{R}_2$-normal. Therefore, all terms in $\epsilon(t'')$ are innermost $\mathcal{R}$-terminating by Lemma 1 (2). This implies that t'' is $\mathcal{R}$-innermost terminating by the above proven claim, contradicting the assumption that t is not innermost $\mathcal{R}$-terminating. Therefore $\mathcal{R}$ is innermost terminating. ■

For instance, $\mathcal{R}_{\mathrm{pg}}$ can be readily proven innermost terminating with Theorem 4. We will present in Appendix A more realistic examples which can be proven (innermost) terminating with the applications of Theorem 1, 2 and 4. We regard these theorems as the major contribution of this paper.

We now present a theorem to demonstrate that the erasure technique can also be directly applied to termination proofs. We first establish a lemma needed later.

Lemma 5. *Let $\succeq$ be an ordering on terms. For multisets S, T_1 and T_2 of terms, if $T_1 \succ^{max} T_2$, then $S \cup T_1 \succ^{(m)} S \cup T_2$.*

Proof The lemma immediately follows from the definition of $\succ^{max}$ and $\succ^{(m)}$. ■

Theorem 5. *Let $\mathcal{S}$ be an erasure TRS in which all the rules are not argument-dropping and $\succeq$ be a reduction ordering which is total on ground terms. Assume $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ such that*

- *$|l| = |r|$ for all rules $l \rightarrow r \in \mathcal{R}_1$, and*
- *$|l| \setminus |r| \succ^{max} |r| \setminus |l|$ for all rules $l \rightarrow r \in \mathcal{R}_2$.*

Then the termination of $\mathcal{R}_1$ implies that of $\mathcal{R}_2$.

Proof (sketch) Assume that $t_1 \to_{\mathcal{R}} t_2/\langle C, l \to r, \sigma\rangle$. It suffices to prove that $|t_1| \succeq^{\textcircled{m}} |t_2|$ if $l \to r \in \mathcal{R}_1$ and $|t_1| \succ^{\textcircled{m}} |t_2|$ if $l \to r \in \mathcal{R}_2$.

If $l \to r \in \mathcal{R}_1$, then $|l| = |r|$, which implies that $|t_1| = |t_2|$. We now assume that $l \to r \in \mathcal{R}_2$. Let $|C| = \{C_1, \ldots, C_k\}$. Since all rules in $\mathcal{S}$ are not argument-dropping, every C_i is a context for $1 \leq i \leq k$. Let us define multisets S, T_1, T_2 of terms as follows.

$$S = \bigcup_{1 \leq i \leq k} \{C_i[s] \mid s \in |t\sigma| \text{ and } t \in |l| \cap |r|\}$$
$$T_1 = \bigcup_{1 \leq i \leq k} \{C_i[s] \mid s \in |t\sigma| \text{ and } t \in |l| \setminus |r|\}$$
$$T_2 = \bigcup_{1 \leq i \leq k} \{C_i[s] \mid s \in |t\sigma| \text{ and } t \in |r| \setminus |l|\}$$

It can be readily proven with Lemma 4 that $T_1 \succ^{max} T_2$ holds. Therefore, Also we can show $|t_1| = S \cup T_1$ and $|t_2| = S \cup T_2$. By Lemma 5, we have $|t_1| \succ^{\textcircled{m}} |t_2|$. ■

Theorem 5 immediately strengthens Theorem 12 (1) in [Zan94], where it is required that f does not occur in l for every $l \to r \in \mathcal{R}$ if the erasure rule (f-p-$(1, \ldots, n)$) is included in $\mathcal{S}$ [3]. Applications of Theorem 5 can be found in Appendix A.

4 Related Work

There is a large number of results in the literature concerning termination proofs for various modular combinations of TRSs. We refer the reader to [Der95] for some clean explanation on many significant results in this area. The general scenario is to prove the termination of $\mathcal{R}_1 \cup \mathcal{R}_2$ for terminating TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$ under some assumption on the relation between $\mathcal{R}_1$ and $\mathcal{R}_2$. We have found that most of the results such as the ones mentioned in [Der95], though interesting, make assumptions about $\mathcal{R}_1$ and $\mathcal{R}_2$ which are too strong for the purpose of verifying the termination of hierarchical combination of $\mathcal{R}_1$ and $\mathcal{R}_2$, sometimes.

We are most interested in the case of hierarchical combination of $\mathcal{R}_1$ and $\mathcal{R}_2$ where the defined function symbols in $\mathcal{R}_1$ are used in $\mathcal{R}_2$ in an essential way since this closely resembles the structure of a functional or logic program. This almost forces us to know the semantics of $\mathcal{R}_1$ to certain extent in order to prove the termination of the combined system. ET is proposed to address the issue in a (very) restricted manner. For instance, the use of the projection rule (*remove*-p-2) in the $\mathcal{R}_{\text{pg}}$ example is simply to test that $remove(x, ys)$ can never return a list of length greater than that of ys. This test succeeds because the generated erasure of $\mathcal{R}_{\text{pg}}$ can be ordered. Let $\mathcal{R}'_{\text{pg}}$ be $\mathcal{R}_{\text{pg}}$ in which the rule $remove(x, nil) \to nil$ is replaced with another rule $remove(x, nil) \to cons(x, nil)$, then the test will fail on $\mathcal{R}'_{\text{pg}}$ since we cannot order $nil \to cons(x, nil)$. Notice that $\mathcal{R}'_{\text{pg}}$ is *not* terminating. This immediately implies that none of the results mentioned in [Der95] can give modular termination proofs for $\mathcal{R}_{\text{pg}}$. If they could, they would also prove this for $\mathcal{R}'_{\text{pg}}$ since $\mathcal{R}_{\text{pg}}$ and $\mathcal{R}'_{\text{pg}}$ exhibit the very same characteristics to them.

The dependency pair approach (DPA) [AG97,AG98], which inspired our work on erasure, deserves special mentioning. We regard ET as a similar idea cast into the general framework of *termination through transformation*. The technical explanation is that, to a large extent, erasure amounts to the use of weak reduction orderings, which are referred as weakly monotonic orderings stable under substitutions in papers on DPA. In general, DPA seems more powerful than ET but it is also (in our opinion) more involved. For instance, DPA uses unification to detect circles of dependency pairs and the set of usable rules, but this is currently unavailable in ET. However, this seems to be a less significant issue so far in our experiment, especially, after we combine ET with the freezing technique [Xi98]. We also plan to incorporate similar ideas into ET if the needs appear. We feel that the most significant advantage of ET over DPA is the availability of nondeterministic erasure rules. Because of the lack of a similar feature,

[3] A strengthened version of this theorem is proven in [MOZ96] which does allow the occurrences of f on the left-hand sides of the rules, but it is nonetheless essentially different from Theorem 5. Please see Example 4 in Appendix A.

DPA is often awkward in handling conditional *if*. For instance, we must order the following rule

$$remove(x, cons(y, ys)) \to if(x = y, remove(x, ys), cons(x, remove(y, ys)))$$

with a weakly monotone ordering if $\mathcal{R}_{\text{pg}}$ is to be proven terminating using DPA. Suppose that we use RPOS as the underlying approach to ordering the rule. We cannot assume $remove \succ cons$ in the precedence relation since this prevents us from strictly ordering the following generated dependency pair $PURGE(cons(x, xs)) > PURGE(remove(x, xs))$. If we map $if(b, x, y)$ to x (y), then the rule $if(false, x, y) \to y$ $(if(true, x, y) \to x)$ cannot be ordered. As a consequence, the *if* function often needs to be "preprocessed" away when DPA is applied because it is difficult to synthesize a weakly monotone ordering based on RPOS or polynomial interpretations in the presence of *if* to order the generated dependency pairs. For instance, the following rules are introduced in the technical report version of [AG97] for handling the purge function example.

$$\begin{aligned} remove(x, cons(y, ys)) &\to ifremove(x = y, x, cons(y, ys)) \\ ifremove(true, x, cons(y, ys)) &\to remove(x, ys) \\ ifremove(false, x, cons(y, ys)) &\to cons(y, remove(x, ys)) \end{aligned}$$

Though the argument is that the introduction of these rules is to forbid rewriting terms under if-branches until the condition is resolved, we feel that this is also a bit unnatural at least since realistic TRSs are seldom formed in such a manner. Notice that the termination of $\mathcal{R}_{\text{pg}}$ is independent of whether we rewrite terms under if-branches or not. It seems straightforward to make use of the weak reduction ordering $\succeq^{max}$ in DPA for handling *if*, and this can elegantly resolve the above issue. We will use some concrete examples to further compare ET or ET plus the freezing technique with DPA in Appendix A.

The use of projection erasure rules bears some resemblance to *distribution elimination* [Zan94], but there are also many significant differences. Although it is clearly possible, there seems no attempt in [Zan94] to construct the ordering $\succeq^{max}$ from a given weak reduction ordering $\succeq$, which we regard as a significant contribution of the paper. Also we mention that the use of an omitting rule (f-o-$(1, \ldots, n)$) in case $\mathcal{A}r(f) = n$ casually relates to *dummy elimination* [Fer96].

5 Conclusion

We have presented a technique named erasure to facilitate the termination and innermost termination proofs, and this technique is inspired by the dependency pair approach in the literature. The erasure technique (ET) is simple to apply and effective in practice, and therefore is reasonable to expect that ET can be combined with other automated approaches to termination proofs for TRS such as freezing [Xi98]. However, we observe in practice that it is even difficult to scale an approach as simple as RPOS. This makes us believe that a more promising direction is to apply ET interactively. In this respect, we have tried ET extensively on various TRSs and the results are encouraging. We present some examples in Appendix A to substantiate this claim.

In general, we are highly motivated to look for approaches to termination proofs for TRSs which are simple and effective. We intend to integrate these approaches into an interactive termination prover for TRSs. The user may be required to interact when applying these approaches but the needed interaction should not be overwhelming. We view this as promising direction to pursue so as to address the following dilemma: too much automation can severely hinder the scalability of a termination proof procedure for TRSs while too little can easily lead to an amount of required interaction which is simply overwhelming for the user. This should be especially clear to those who have used interactive theorem provers such as PVS [ORR+96] or Isabelle [Law94] for proving the termination of recursively defined functions.

References

[AG97] Thomas Arts and Jürgen Giesl. Proving innermost normalisation automatically. In Hubert Comon, editor, *Proceedings of the 8th Conference on Rewriting Techniques and Applications*, pages 157–171.

Springer-Verlag LNCS 1232, 1997. An extended version is available as Technical Report IBN 96/39. Technische Hochschule Darmstadt.

[AG98] Thomas Arts and Jürgen Giesl. Modularity of termination using dependency pairs. In Tobias Nipkow, editor, *Proceedings of the 9th Conference on Rewriting Techniques and Applications*, pages 226–240. Springer-Verlag LNCS 1379, 1998.

[AZ95] Thomas Arts and Hans Zantema. Termination of logical programs using semantic unification. In *Proceedings of the 5th International Workshop on Logical Program Synthesis and Transformation*, pages 219–233, Utrecht, 1995. Springer-Verlag LNCS 1048.

[BL87] Ahkem BenCherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *SCP*, 9(2):137–160, 1987.

[BL90] Francoise Bellegarde and Pierre Lescanne. Termination by completion. *Applicable Algebra in Engineering, Communication and Computing*, 1:79–96, 1990.

[CHR92] P.-L. Curien, T. Hardin, and A. Ríos. Strong normalization of substitutions. In I. M. Havel and V. Koubek, editors, *Proceedings of Mathematical Foundations of Computer Science*, pages 209–217. Springer-Verlag LNCS 629, 1992.

[Der82] Nachum Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[Der95] Nachum Dershowitz. Hierarchical termination. In *Proceedings of the 4th International Workshop on Conditional and Typed Rewriting Systems*, pages 89–105. Springer-Verlag LNCS 968, 1995.

[DJ91] Nachum Dershowitz and Jean-Pierre Jouannaud. Notations for rewriting. *EATCS*, 43:162–172, 1991.

[Fer96] Maria Ferreira. Dummy elimination in equational rewriting. In Harald Ganzinger, editor, *Proceedings of the 7th Conference on Rewriting Techniques and Applications*, pages 78–92. Springer-Verlag LNCS 1103, 1996.

[Gra95] Bernhard Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24(1/2):2–23, 1995.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[KL80] Sam Kamin and Jean-Jacques Lévy. Attempts for generalizing the recursive path orderings. Unpublished manuscript, February 1980.

[Lan79] Dallas Lankford. On proving term rewriting systems are noetherian. Technical Report Memo MTP-3, Louisiana Tech. University, 1979.

[Law94] Paul Lawrence. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[MOZ96] Aart Middeldorp, Hitoshi Ohsaki, and Hans Zantema. Transforming termination by self-labelling. In *Proceedings of 13th International Conference on Automated Deduction*, pages 373–387, New Brunswick, July/August 1996. Springer-Verlag LNCS 1104.

[MTHM97] Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1997.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification, CAV '96*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag LNCS 1102.

[Pla78] David Plaisted. A recursively defined ordering for proving termination of term rewriting systems. Technical Report UIUC DCS-R-78-943, Univ. of Illinois at Urbana-Champaign, 1978.

[Ste95a] Joachim Steinbach. Automatic termination proofs with transformation orderings. In *Proceedings of the 6th Conference on Rewriting Techniques and Applications*, pages 11–25. Springer-Verlag LNCS 914, 1995.

[Ste95b] Joachim Steinbach. Simplification orderings: History of results. *Fundamenta Informaticae*, 24:47–87, 1995.

[SX98] Joachim Steinbach and Hongwei Xi. Freezing – termination proofs for classical, context sensitive and innermost rewriting. Technical report, Technische Universität München, 1998.

[Xi98] Hongwei Xi. Towards automated termination proofs through "freezing". In *9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 271–285, March–April 1998.

[Zan94] Hans Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.

[Zan95] Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

A Examples

Example 1. We often combine ET with the freezing technique [Xi98] in practice. Let $\mathcal{R}_{\text{fact}}$ be the following TRS [KL80].

$$\begin{array}{rr} p(s(0)) \to 0 & fact(0) \to s(0) \\ p(s(s(x))) \to s(p(s(x))) & fact(s(x)) \to s(x) * fact(p(s(x))) \end{array}$$

The following $\mathcal{R}^1_{\text{fact}}$ is a $(p, s, \underline{ps}, 1)$-frozen version of $\mathcal{R}_{\text{fact}}$, and therefore the termination of $\mathcal{R}^1_{\text{fact}}$ implies that of $\mathcal{R}_{\text{fact}}$.

$$\begin{array}{lll} (1) & p(s(0)) \to 0 & (4) \quad \underline{ps}(s(x)) \to s(\underline{ps}(x)) \\ (2) & \underline{ps}(0) \to 0 & (5) \quad fact(0) \to s(0) \\ (3) & p(s(s(x))) \to s(\underline{ps}(x)) & (6) \quad fact(s(x)) \to s(x) * fact(\underline{ps}(x)) \end{array}$$

The following $\mathcal{R}^2_{\text{fact}}$ is the $\mathcal{S}$-erasure of $\mathcal{R}^1_{\text{fact}}$ for $\mathcal{S} = \{\underline{ps}\text{-p-1}\}$.

$$\begin{array}{lll} (1') & p(s(0)) \to 0 & (4') \quad s(x) \to s(x) \\ (2') & 0 \to 0 & (5') \quad fact(0) \to s(0) \\ (3') & p(s(s(x))) \to s(x) & (6') \quad fact(s(x)) \to s(x) * fact(x) \end{array}$$

Under the RPO with the precedence $fact \succ *$, rules (2') and (4') can be ordered and the rest of the rules can be strictly ordered. Since the TRS consisting of rules (2) and (4) is obviously terminating, the termination of $\mathcal{R}^1_{\text{fact}}$ follows from Theorem 1. Therefore, $\mathcal{R}_{\text{fact}}$ is terminating by a theorem on the freezing technique.

If we apply DPA to $\mathcal{R}_{\text{fact}}$, the following dependency pair is generated.

$$FACT(s(x)) > FACT(p(s(x)))$$

It is unclear how this can be strictly ordered since we cannot project away the argument of p because of the existence of the rule $p(s(s(x))) \to s(p(s(x)))$. If one argues that this example is too contrived, then the following example exhibits the same characteristics.

Example 2. In the following TRS $\mathcal{R}_{\log}$, $h(n) = \lfloor n/2 \rfloor$ for every natural number n, and $\log(n) = 1 + \log_2(n)$ for $n > 0$.

$$\begin{array}{rr} h(0) \to 0 & \log(0) \to 0 \\ h(s(0)) \to 0 & \log(s(x)) \to s(\log(h(s(x)))) \\ h(s(s(x))) \to s(h(x)) & \end{array}$$

The last rule is self-embedding, and therefore the termination of this TRS cannot be proven with a simplification ordering. We form an $(h, s, 1, \underline{hs})$-frozen version $\mathcal{R}^1_{\log}$ of $\mathcal{R}_{\log}$ as follows.

$$\begin{array}{lll} (1) & h(0) \to 0 & (5) \quad \underline{hs}(s(x)) \to s(h(x)) \\ (2) & h(s(0)) \to 0 & (6) \quad \log(0) \to 0 \\ (3) & \underline{hs}(0) \to 0 & (7) \quad \log(s(x)) \to s(\log(\underline{hs}(x))) \\ (4) & h(s(s(x))) \to s(h(x)) & \end{array}$$

We can prove the termination of $\mathcal{R}^1_{\log}$ by forming its $\mathcal{S}$-erasure for $\mathcal{S} = \{h\text{-p-1}, \underline{hs}\text{-p-1}\}$. This then implies the termination of $\mathcal{R}_{\log}$. We omit the details that are straightforward to fill in. Notice that this example can not handled by DPA for the same reason as explained in the previous example.

In general, we intend to apply various transformations for proving the (innermost) termination of a TRS $\mathcal{R}$. We generate a chain of TRSs $\mathcal{R} = \mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n$ such that the (innermost) termination of $\mathcal{R}_{i+1}$ implies that of $\mathcal{R}_i$ for $1 \leq i < n$ and the (innermost) termination of $\mathcal{R}_n$ can be proven with some basic approach such as RPOS or polynomial interpretations. The problem with DPA is that it generates a set of dependency pairs rather than a TRS, and therefore it is difficult to be combined with other transformational approaches.

Example 3. The following TRS $\mathcal{R}_{\text{qs}}$ defines a quicksort function on lists. Let $\mathcal{R}_1$ consist of all these rules except the last 2, and $\mathcal{R}_2$ consist of the last 2 rules. Then $\mathcal{R}_{\text{qs}}$ is the hierarchical combination of $\mathcal{R}_1$ and $\mathcal{R}_2$.

$$
\begin{array}{ll}
(1) & \mathit{if}(\mathit{true}, x, y) \to x \\
(2) & \mathit{if}(\mathit{false}, x, y) \to y \\
(3) & 0 \le x \to \mathit{true} \\
(4) & s(x) \le 0 \to \mathit{false} \\
(5) & s(x) \le s(y) \to x \le y \\
(6) & \mathit{gte}(x, []) \to [] \\
(7) & \mathit{gte}(x, y :: ys) \to \mathit{if}(x \le y, y :: \mathit{gte}(x, ys), \mathit{gte}(x, ys)) \\
(8) & \mathit{lt}(x, []) \to [] \\
(9) & \mathit{lt}(x, y :: ys) \to \mathit{if}(x \le y, \mathit{lt}(x, ys), y :: \mathit{lt}(x, ys)) \\
(10) & [] \mathbin{@} ys \to ys \\
(11) & (x :: xs) \mathbin{@} ys \to x :: (xs \mathbin{@} ys) \\
(12) & \mathit{quicksort}([]) \to [] \\
(13) & \mathit{quicksort}(x :: xs) \to \mathit{quicksort}(\mathit{lt}(x, xs)) \mathbin{@} [x] \mathbin{@} \mathit{quicksort}(\mathit{gte}(x, xs))
\end{array}
$$

The following is the $\mathcal{S}$-erasure of $\mathcal{R}$ for $\mathcal{S} = \{(\mathit{if}\text{-p-}(2,3)), (\mathit{gte}\text{-p-}2), (\mathit{lt}\text{-p-}2)\}$.

$$
\begin{array}{ll}
(1') & \{x, y\} \to x \\
(2') & \{x, y\} \to y \\
(3') & 0 \le x \to \mathit{true} \\
(4') & s(x) \le 0 \to \mathit{false} \\
(5') & s(x) \le s(y) \to x \le y \\
(6') & [] \to [] \\
(7') & y :: ys \to \{y :: ys, ys\} \\
(8') & [] \to [] \\
(9') & y :: ys \to \{ys, y :: ys\} \\
(10') & [] \mathbin{@} ys \to ys \\
(11') & (x :: xs) \mathbin{@} ys \to x :: (xs \mathbin{@} ys) \\
(12') & \mathit{quicksort}([]) \to [] \\
(13') & \mathit{quicksort}(x :: xs) \to \mathit{quicksort}(xs) \mathbin{@} [x] \mathbin{@} \mathit{quicksort}(xs)
\end{array}
$$

Under the RPO with precedence $\mathit{quicksort} \succ @, \le \succ \mathit{true}, \mathit{false}$, all rule erasures in $\mathcal{R}_1$ can be ordered and all rule erasures in $\mathcal{R}_2$ can be strictly ordered. By Theorem 4, the innermost termination of $\mathcal{R}_{\text{qs}}$ follows from that of $\mathcal{R}_1$. It can be readily proven with a RPO that $\mathcal{R}_1$ is (innermost) terminating, and therefore $\mathcal{R}_{\text{qs}}$ is innermost terminating. This implies that $\mathcal{R}_{\text{qs}}$ is terminating since it is non-overlapping.

A similar example also appears in the technical report version of [AG97], but *if* is "preprocessed" away. The termination of that example can be readily proven with Theorem 2.

Example 4. Let $\mathcal{R}_1$ be the following TRS.

$$
\begin{array}{ll}
(1) & f(\emptyset) \to \emptyset \\
(2) & f(\mathit{branch}(\emptyset, x)) \to \mathit{branch}(\emptyset, f(x)) \\
(3) & f(\mathit{branch}(\mathit{branch}(x, y), z)) \to f(\mathit{branch}(x, \mathit{branch}(y, z))) \\
(4) & g(\emptyset) \to \emptyset \\
(5) & g(\mathit{branch}(x, \emptyset)) \to \mathit{branch}(\emptyset, g(x)) \\
(6) & g(\mathit{branch}(x, \mathit{branch}(y, z))) \to g(\mathit{branch}(\mathit{branch}(x, y), z))
\end{array}
$$

The following is the $\mathcal{S}$-erasure of $\mathcal{R}_1$ for $\mathcal{S} = \{branch\text{-p-}(1,2)\}$.

$$\begin{array}{ll}
(1') & f(\emptyset) \to \emptyset \\
(2') & \{f(\emptyset), f(x)\} \to \{\emptyset, f(x)\} \\
(3') & \{f(x), f(y), f(z)\} \to \{f(x), f(y), f(z)\} \\
(4') & g(\emptyset) \to \emptyset \\
(5') & \{g(x), g(\emptyset)\} \to \{\emptyset, g(x)\} \\
(6') & \{g(x), g(y), g(z)\} \to \{g(x), g(y), g(z)\}
\end{array}$$

By Theorem 5, the termination of $\mathcal{R}_1$ follows from the termination of $\mathcal{R}_2 = \{(3), (6)\}$. We now construct a TRS $\mathcal{R}_3$ below, which is an $(f, branch, 1, \underline{fbranch})$-frozen version of $\mathcal{R}_2$.

$$\begin{array}{c}
\underline{fbranch}(branch(x,y),z) \to \underline{fbranch}(x, branch(y,z)) \\
g(\underline{branch}(x, branch(y,z))) \to g(branch(branch(x,y),z))
\end{array}$$

The termination of $\mathcal{R}_3$ is easily proven with a RPOS, and therefore, $\mathcal{R}_1$ is terminating. We point out that it would be greatly involved (though possible) if we applied the freezing technique to $\mathcal{R}_1$ directly.

Notice that Theorem 12 [Zan94] cannot be applied to this example since f has occurrences on the left-hand sides of the rules. The strengthened version of this theorem in [MOZ96] cannot handle this example, either.

Example 5. The termination of the following TRS σ_0 describes the process of substitution in combinatory logic, and the proof for the termination of σ_0 in [CHR92] is involved. Some simplified proofs have been given in [Zan94,Zan95].

$$\begin{array}{ll}
(1) & \lambda(x) \circ y \to \lambda(x \circ (1 \cdot (y \circ \uparrow))) \\
(2) & (x \cdot y) \circ z \to (x \circ z) \cdot (y \circ z) \\
(3) & (x \circ y) \circ z \to x \circ (y \circ z) \\
(4) & id \circ x \to x \\
(5) & 1 \circ id \to 1 \\
(6) & 1 \circ (x \cdot y) \to x \\
(7) & \uparrow \circ (x \cdot y) \to y
\end{array}$$

The following is the $\mathcal{S}$-erasure of σ_0 for $\mathcal{S} = \{(\cdot\text{-p-}(1,2))\}$.

$$\begin{array}{ll}
(1') & \lambda(x) \circ y \to \{\lambda(x \circ 1), \lambda(x \circ (y \circ \uparrow))\} \\
(2') & \{x \circ z, y \circ z\} \to \{x \circ z, y \circ z\} \\
(3') & (x \circ y) \circ z \to x \circ (y \circ z) \\
(4') & id \circ x \to x \\
(5') & 1 \circ id \to 1 \\
(6') & \{1 \circ x, 1 \circ y\} \to x \\
(7') & \{\uparrow \circ x, \uparrow \circ y\} \to y
\end{array}$$

As shown in [Zan94], all the rule erasures except the second one can be strictly ordered under a total ordering. By Theorem 5, the termination of σ_0 follows from the termination of the TRS consisting of the rule $(x \cdot y) \circ z \to (x \circ z) \cdot (y \circ z)$, which is obvious. Notice that the distribution elimination technique [Zan94] cannot be directly applied to this example because of the occurrences of $\cdot$ on the left-hand sides of some rules. If we replace the last rule in σ_0 with $\uparrow \circ (x \cdot y) \to y \cdot x$, then the strategy used in [Zan94] would no longer work but Theorem 5 could still be applied.

Example 6. The following example is adopted from the technical report version of [AG97], where it is formed as a variation of an algorithm in [?]. The purpose of the function $rename(x, y, t)$ is to replace

every free occurrence of the variable x in the term t with the variable y.

$$
\begin{array}{ll}
(1) & \mathit{true} \wedge y \to y \\
(2) & \mathit{false} \wedge y \to \mathit{false} \\
(3) & [] = [] \to \mathit{true} \\
(4) & (x :: xs) = [] \to \mathit{false} \\
(5) & [] = (y :: ys) \to \mathit{false} \\
(6) & (x :: xs) = (y :: ys) \to (x = y) \wedge (xs = ys) \\
(7) & \mathit{var}(xs) = \mathit{var}(ys) \to xs = ys \\
(8) & \mathit{var}(xs) = \mathit{apply}(s,t) \to \mathit{false} \\
(9) & \mathit{var}(xs) = \mathit{lambda}(x,s) \to \mathit{false} \\
(10) & \mathit{apply}(s,t) = \mathit{var}(ys) \to \mathit{false} \\
(11) & \mathit{apply}(s,t) = \mathit{apply}(u,v) \to (s = u) \wedge (t = v) \\
(12) & \mathit{apply}(s,t) = \mathit{lambda}(x,u) \to \mathit{false} \\
(13) & \mathit{lambda}(x,s) = \mathit{var}(ys) \to \mathit{false} \\
(14) & \mathit{lambda}(x,s) = \mathit{apply}(u,v) \to \mathit{false} \\
(15) & \mathit{lambda}(x,s) = \mathit{lambda}(y,t) \to (x = y) \wedge (s = t) \\
(16) & \mathit{if}(\mathit{true}, \mathit{var}(xs), \mathit{var}(ys)) \to \mathit{var}(xs) \\
(17) & \mathit{if}(\mathit{false}, \mathit{var}(xs), \mathit{var}(ys)) \to \mathit{var}(ys) \\
(18) & \mathit{rename}(\mathit{var}(xs), \mathit{var}(ys), \mathit{var}(zs)) \to \mathit{if}(xs = zs, \mathit{var}(ys), \mathit{var}(zs)) \\
(19) & \mathit{rename}(x, y, \mathit{apply}(s,t)) \to \mathit{apply}(\mathit{rename}(x,y,s), \mathit{rename}(x,y,t)) \\
(20) & \mathit{rename}(x, y, \mathit{lambda}(z,t)) \to \mathit{lambda}(\bullet, \mathit{rename}(x, y, \mathit{rename}(z, \bullet, t)))
\end{array}
$$

Note that $\bullet$ in rule (20) stands for $\mathit{var}([x, y, \mathit{lambda}(z,t)])$. Let $\mathcal{R}_1$ consist of the first 17 rules and $\mathcal{R}_2$ consist of the rest of rules. Then $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ is a hierachical combination of $\mathcal{R}_1$ and $\mathcal{R}_2$. Clearly, $\mathcal{R}_1$ can be proven terminating with some RPO. We form the following $\mathcal{S}$-erasure $|\mathcal{R}|$ of $\mathcal{R}$ for $\mathcal{S} = \{(\wedge\text{-p-2}), (=\text{-o-}(1,2)), (\mathit{var}\text{-o-}1), (\mathit{if}\text{-o-}(1,2,3)), (\mathit{rename}\text{-p-}3), (\mathit{lambda}\text{-o-}1)\}$.

$$
\begin{array}{ll}
(1') & y \to y \\
(2') & y \to \mathit{false} \\
(3') & = \to \mathit{true} \\
(4') & = \to \mathit{false} \\
(5') & = \to \mathit{false} \\
(6') & = \to = \\
(7') & = \to = \\
(8') & = \to \mathit{false} \\
(9') & = \to \mathit{false} \\
(10') & = \to \mathit{false} \\
(11') & = \to = \\
(12') & = \to \mathit{false} \\
(13') & = \to \mathit{false} \\
(14') & = \to \mathit{false} \\
(15') & = \to = \\
(16') & \mathit{if} \to \mathit{var} \\
(17') & \mathit{if} \to \mathit{var} \\
(18') & \mathit{var} \to \mathit{if} \\
(19') & \mathit{apply}(s,t) \to \mathit{apply}(s,t) \\
(20') & \mathit{lambda}(t) \to \mathit{lambda}(t)
\end{array}
$$

It can be readily verified that $|\mathcal{R}|$ is $\mathcal{R}_2$-conservative. Under the RPO with the precedence relation $\mathit{true} \approx \mathit{false} \approx = \approx \mathit{var} = \mathit{if}$, all the rules can be ordered. Note that rule (2') is ordered because we can require that *false* be a constant with the lowest precedence. Let $\succeq_1$ denote this RPO. We can then form an ordering $\succ_2$ with the precedence $\mathit{rename} \succ \mathit{apply}, \mathit{lambda}$ as described in Definition 7. Then the right

side of rule (18) is $\mathcal{R}_2$-skeleton normal, and both rules (19) and (20) can be ordered under $\succ_2$. Therefore, $\mathcal{R}$ is terminating by Theorem 3.

Pacific Software Research Center
Department of Computer Science and Engineering
(503) 690-1151
Internet: *pacsoft@cse.ogi.edu*
World Wide Web: *http://www.cse.ogi.edu/PacSoft/*

Cover: The chart on the cover tracks the problem severity factor (PSF) of each of PacSoft's program transformation tools during the software development cycle. Low PSFs indicate that detected defects in the software have been fixed. This chart was constructed by Alexei Kotov, a graduate student for the Pacific Software Research Center.

Oregon Graduate Institute of Science & Technology
P.O. Box 91000
Portland, OR 97291-1000